

版权注意事项：1、书籍版权归著者和出版社所有；
2、本PDF仅用于个人获取知识，进行私底下知识交流；
3、PDF获得者不得在互联网以任何目的进行传播；
如有需要，请尽量购买正版实体书！支持书籍作者！！



揭秘Java虚拟机

JVM设计原理与实现

封亚飞 著



中国工信出版集团



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

内容简介

本书全面系统地介绍了Java虚拟机（JVM）的设计原理与实现。全书共分10章，第1章介绍Java语言的发展概况；第2章介绍Java虚拟机（JVM）的体系结构；第3章介绍Java虚拟机（JVM）的启动过程；第4章介绍Java虚拟机（JVM）的类加载过程；第5章介绍Java虚拟机（JVM）的字节码解释器；第6章介绍Java虚拟机（JVM）的垃圾回收机制；第7章介绍Java虚拟机（JVM）的线程模型；第8章介绍Java虚拟机（JVM）的本地方法接口；第9章介绍Java虚拟机（JVM）的调试接口；第10章介绍Java虚拟机（JVM）的国际化支持。本书可作为高等院校计算机专业及相关专业的教材，也可供从事Java开发的工程技术人员参考。



揭秘Java虚拟机

JVM设计原理与实现

封亚飞 著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

本书从源码角度解读HotSpot的内部实现机制, 本版本主要包含三大部分——JVM数据结构设计与实现、执行引擎机制及内存分配模型。

数据结构部分包括Java字节码文件格式、常量池解析、字段解析、方法解析。每一部分都给出详细的源码实现分析, 例如字段解析一章, 从源码层面详细分析了Java字段重排、字段继承等关键机制。再如方法解析一章, 给出了Java多态特性在源码层面的实现方式。本书通过直接对源代码的分析, 从根本上梳理和澄清Java领域中的关键概念和机制。

执行引擎部分包括Java方法调用机制、栈帧创建机制、指令集架构与解释器实现机制。这一话题是全书技术含量最高的部分, 需要读者具备一定的汇编基础。不过千万不要被“汇编”这个词给吓着, 其实在作者看来, 汇编相比于高级语言而言, 语法非常简单, 语义也十分清晰。执行引擎部分重点描述Java源代码如何转换为字节码, 又如何从字节码转换为机器指令从而能够被物理CPU所执行的技术实现。同时详细分析了Java函数堆栈的创建全过程, 在源码分析的过程中, 带领读者从本质上理解到底什么是Java函数堆栈和栈帧, 以及栈帧内部的详细结构。

内存分配部分主要包括类型创建与加载、对象实例创建与内存分配, 例如new关键字的工作机制, import关键字的作用, 再如java.lang.ClassLoader.loadClass()接口的本地实现机制。

本书并不是简单地分析源码实现, 而是在描述HotSpot内部实现机制的同时, 分析了HotSpot如此这般实现的技术必然性。读者在阅读本书的过程中, 将会在很多地方看到作者本人的这种思考。

未经许可, 不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有, 侵权必究。

图书在版编目(CIP)数据

揭秘 Java 虚拟机: JVM 设计原理与实现 / 封亚飞著. —北京: 电子工业出版社, 2017.7

ISBN 978-7-121-31541-1

I. ①揭… II. ①封… III. ①JAVA 语言—程序设计 IV. ①TP312.8

中国版本图书馆 CIP 数据核字(2017)第 101824 号

策划编辑: 刘 皎

责任编辑: 郑柳洁

特约编辑: 梁卫红

印 刷: 北京京科印刷有限公司

装 订: 三河市良远印务有限公司

出版发行: 电子工业出版社

北京市海淀区万寿路 173 信箱 邮编: 100036

开 本: 787×980 1/16 印张: 42.25 字数: 942 千字

版 次: 2017 年 7 月第 1 版

印 次: 2017 年 7 月第 1 次印刷

定 价: 129.00 元

凡所购买电子工业出版社图书有缺损问题, 请向购买书店调换。若书店售缺, 请与本社发行部联系, 联系及邮购电话: (010) 88254888, 88258888。

质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式: 010-51260888-819, faq@phei.com.cn。

推荐序

从 Java 诞生至今已有二十余年，基于虚拟机的技术屏蔽了底层环境的差异，“一次编译，随处运行”的思想促进了整个 IT 上层技术应用产生了翻天覆地的变化。Java 作为服务端应用语言的首选，确实大大降低了学习和应用的门槛。现实生活中，绝大多数 Java 程序员对于虚拟机的原理和实现了解并不深入，也似乎并不那么关心。而随着互联网的极速发展，现在的 Java 服务端应用需要应对极高的并发访问和大量的数据交互，从机制和设计原理上了解虚拟机的核心原理和实现细节显然能够帮助 Java 程序员编写出更高效优质的代码。

虽然市面上从 Java 使用者角度介绍虚拟机的书也有不少佳作，但一般较为宽泛，尤其在谈及虚拟机如何运行、处理的细节时总有些浅尝辄止的遗憾。而作者凭借深厚的 C 与 Java 技术功底以及多年对于 JVM 的深入研究编写的这本书，真正从虚拟机指令执行处理层面，结合 JVM 规范的设计原理，完整和详尽地阐述了 Java 虚拟机在处理类、方法和代码时的设计和实现细节。书中大量的代码和指令细节能够让程序员更加直接地理解相关原理。

这是一本优秀的技术工具书，可以让阅读者更加深刻地理解虚拟机的原理和处理细节，值得每一位具有极客精神、追求细节的优秀程序员反复阅读和收藏。

菜鸟平台技术部 陌铭

前言

文明需要创造，也需要传承。JVM 作为一款虚拟机，本身便是技术之集大成者，里面包含方方面面的底层技术知识。抛开如今 Java 如日中天之态势不说，纯粹从技术层面看，JVM 也值得广大技术爱好者深入研究。可以说，从最新的硬件特性，到最新的软件技术，只要技术被证明是成熟的，都会在 JVM 里面见到其踪影。JDK 的每一次更新，从内部到核心类库，JVM 都会及时引入这些最新的技术或者算法，这便是技术传承意义之所在。随着云计算、大数据、人工智能等最新技术的发展，Java 技术生态圈也日益庞大，JVM 与底层平台以及与其他编程语言和技术的交互、交织日益深入，这些都离不开对 JVM 内部机制的深入理解。如果说以前在中间件与框架领域的大展身手，依靠的是 Java 语言层面的特性和技术，那么以后越来越多的技术红利将会因 JVM 层面之创新而得以显现。

被真相所蒙蔽，是一件痛苦的事。我们在一个被层层封装的世界里进行开发和设计，操作系统、各种中间件与框架，将底层世界隐藏得结结实实。我们一方面享受着高级编程语言所带来的高效、稳定、快速的开发体验，然而另一方面，却又如同行走于黑暗之中。我们不知道路的下面是否有坑，即使有坑，可能也不知道如何排除。Java 的很多概念和技术，很多时候由于我们对底层机制的不了解，而让我们感到十分高深莫测，无法知其全貌。这种感觉非常痛苦，尤其是技术修炼到一定阶段的时候。

纸上得来终觉浅，绝知此事要躬行。即使从 Java 语言层面下探到 JVM 层面，但是若只囿于对 JVM 机制理论和概念上的理解，很多时候仍然觉得缺乏那种大彻大悟之感。计算机作为一门科学，与其他的科学领域一样，不仅需要对其理论的理解，也需要能够去实证。例如爱因斯坦的相对论十分高深，但是通过对引力波和红移的观测，其变得形象和生动起来。Java 的部分概念经过“口口相传”，似有过于夸大其技术神秘性之嫌，让人望而生畏。例如，与 `volatile` 关键字相关的内存可见性、指令乱序等概念，给人无比博大深奥的印象，但是如能抛开概念，直接看底层实现机制，并辅以具体的实验论证，则会形成深刻而彻底的认知。其实，这世界本来就很简单。在可观测的实验结果与可理解的底层机制面前，一切浮夸的概念都自然会现出原形。

因此，采用自底而上的技术研究之道，相比自顶而下的办法，便多了更多窥透本质的自信和平实。同一个底层概念，在不同的高级编程语言里，在概念、叫法上很少能够保持一致。采用自底而上的探索方法，能够揭开各种深奥概念的神秘面纱，还原一个清明简洁的世界。自然理解曲线也不会有大起大落。

研究 JVM 的过程，就是与大师们进行精神沟通和心灵交流的过程，虽然过程会比较痛苦。研究诸如 Linux、JVM 这样的底层程序，你能学习到大师级的理念，更能够见识到经无数牛人反复锤炼后的技术。天长日久的耳濡目染，终有一天你也会成为大师，你也会拥有大师级的眼光，你也会拥有开阔的胸怀。如同音乐家李健，人们如此喜欢他，并不仅仅是因为他歌唱得好，更多的是因为气质。而这种气质来自于博览群书，来自于对艺术的长久修炼。计算机从某种程度上而言，也是一门艺术，工程师和程序员们要想进化，对计算机艺术的修炼必不可少。与大师进行精神沟通，不仅能够修炼到计算机的艺术，更能直接感受并养成大师身上所具备的气质。

我不知道 Java 还能走多远，未来是否会被淘汰，但你不能因此就否定研究 JVM 的意义。JVM 作为一款虚拟机，各种底层技术和理论都有涉及，若你能研究透彻，则能一通百通。例如，本人在研究过程中，也翻阅了诸如 Python、JavaScript 等高级面向对象语言虚拟机的机制，发现它们内部的整体思路都相差不大。同时，JVM 本身在运行期干了一部分 C 或 C++ 语言编译器所干的事，例如符号解析、链接、面向对象机制的实现等，通过对这些机制的分析，从来没有研究过 C/C++ 编译器原理的我，基本也能够猜出 C/C++ 编译器可能的实现方式，后来翻阅了相关资料，果不其然。理解编译与虚拟机的实现机制是一方面，另一方面，通过深挖 JDK 核心类库的内部实现，则能够深刻理解线程、并发、I/O 等比较高深的技术内幕。例如 Java NIO，何谓 VMA？何谓内核映射？若想真正彻底理解这些概念，不从底层入手，恐怕很难有一个具象化的认知。总之，研究 JVM，是一件非常能够提升开发者内功的事情，未来无论出现什么样的新语言、新技术、新概念，你总是能够不被表面的东西所迷惑，而是能够透过层层封装，看清事物的本质，你总是能够以极低的学习成本，迅速理解新的东西。从一个更为广阔的视角，使用发散的思维去看，不一定非要研究 JVM 才能有很大收获，研究其他技术的底层，会有异曲同工之妙。而我只不过恰好生在了这个年代，这个 Java 语言大行其道的年代，所以就恰好对其做了一个比较深入的研究而已。工具有时空疆界，而技术思想则没有，其总能穿越千万年的时空，无限延伸。

JVM 涉及的知识面十分广阔，因此限于篇幅，本书并未覆盖 JVM 的全部内容。总体而言，本书重点描述了 JVM 从启动开始到完成函数执行的详细机制，读完本书，相信你一定能够明白 JVM 执行 Java 程序的底层机制，能够明白 JVM 将 Java 语言一步步转换为 CPU 可执行的机器码的内部机制，以及为此而制定的各种规范的实现之道，例如 oop-klass 模型、堆栈分配模型、类加载模型等。

本书作为笔者本人的处女作，前后写了有两年之久。之所以写这么久，一方面是因为 JVM

本身涉及大量的知识，另一方面则是笔者本人在写作过程中，力求对每一个知识点都做实验进行验证，避免因为笔者的错误理解而误导了别人。同时，在这个过程中，笔者不仅仅满足于读懂 JVM 的源代码，也不仅仅满足于通过实验去验证各个技术点，笔者花了更多的时间在思考 JVM 各种技术选择的必然性。换言之，在具体的硬件和操作系统的约束之下，在 JVM “write once, run anywhere” 这一思路设定下，JVM 内部的技术实现机制是确定的，别无他法。例如，JVM 的 GC 机制便是一种技术必然性选择。每一次对这种技术必然性之思考，就好像与 JVM 的作者大牛们进行了一次心灵交流。

我问大牛：JVM 的堆栈结构为什么要有操作数栈和局部变量表？

大牛回答：因为……。

大牛其实并没有回答我，所有的一切都需要笔者自己去想明白。等想明白了之后，才有种真正看透事物本质的快感。

希望本书的读者也能够跟随本书，一起去进行这样的思考。

正是因为对“知其所以然”之追求，所以本书的写作过程是漫长的。在此，要特别感谢我的老婆金艳和我的儿子佑佑，很多个周末我都没能抽出时间陪伴他们。当我开始打算写作本书时，我的孩子还在襁褓之中。而等我写完本书，孩子已经开始上小班了。欠缺的太多！

JVM 所涉及的知识面既广且深，而个人所知毕竟有限，书中定有错误之处，若有发现，请发送邮件至：chaomengyuexiang@126.com。

在写作本书的过程中，遇到了很多技术障碍，有很多技术障碍都是在查阅了 R 大以及阿里技术专家三红、寒泉子等人的文章后才得以攻克，在此表示感谢！向这些大牛致敬！

同时，也要感谢我的领导和同事所给予的大力支持，尤其要感谢菲青、陌铭、祝幽、兰博等人的鼓励，同时你们也是我学习的榜样！

目录

第 1 章 Java 虚拟机概述	1
1.1 从机器语言到 Java——詹爷，你好	1
1.2 兼容的选择：一场生产力的革命	6
1.3 中间语言翻译	10
1.3.1 从中间语言翻译到机器码	11
1.3.2 通过 C 程序翻译	11
1.3.3 直接翻译为机器码	13
1.3.4 本地编译	16
1.4 神奇的指令	18
1.4.1 常见汇编指令	20
1.4.2 JVM 指令	21
1.5 本章总结	24
第 2 章 Java 执行引擎工作原理：方法调用	25
2.1 方法调用	26
2.1.1 真实的机器调用	26
2.1.2 C 语言函数调用	41
2.2 JVM 的函数调用机制	47
2.3 函数指针	53
2.4 CallStub 函数指针定义	60
2.5 _call_stub_entry 例程	72

2.6	本章总结	115
第 3 章	Java 数据结构与面向对象	117
3.1	从 Java 算法到数据结构	118
3.2	数据类型简史	122
3.3	Java 数据结构之偶然性	129
3.4	Java 类型识别	132
3.4.1	class 字节码概述	133
3.4.2	魔数与 JVM 内部的 int 类型	136
3.4.3	常量池与 JVM 内部对象模型	137
3.5	大端与小端	143
3.5.1	大端和小端的概念	146
3.5.2	大小端产生的本质原因	148
3.5.3	大小端验证	149
3.5.4	大端和小端产生的场景	151
3.5.5	如何解决字节序反转	154
3.5.6	大小端问题的避免	156
3.5.7	JVM 对字节码文件的大小端处理	156
3.6	本章总结	159
第 4 章	Java 字节码实战	161
4.1	字节码格式初探	161
4.1.1	准备测试用例	162
4.1.2	使用 javap 命令分析字节码文件	162
4.1.3	查看字节码二进制	165
4.2	魔数与版本	166
4.2.1	魔数	168
4.2.2	版本号	168
4.3	常量池	169
4.3.1	常量池的基本结构	169
4.3.2	JVM 所定义的 11 种常量	170
4.3.3	常量池元素的复合结构	170
4.3.4	常量池的结束位置	172

4.3.5	常量池元素总数量.....	172
4.3.6	第一个常量池元素.....	173
4.3.7	第二个常量池元素.....	174
4.3.8	父类常量.....	174
4.3.9	变量型常量池元素.....	175
4.4	访问标识与继承信息.....	177
4.4.1	access_flags.....	177
4.4.2	this_class.....	178
4.4.3	super_class.....	179
4.4.4	interface.....	179
4.5	字段信息.....	180
4.5.1	fields_count.....	180
4.5.2	field_info fields[fields_count].....	181
4.6	方法信息.....	185
4.6.1	methods_count.....	185
4.6.2	method_info methods[methods_count].....	185
4.7	本章回顾.....	205
第 5 章	常量池解析.....	206
5.1	常量池内存分配.....	208
5.1.1	常量池内存分配总体链路.....	209
5.1.2	内存分配.....	215
5.1.3	初始化内存.....	223
5.2	oop-klass 模型.....	224
5.2.1	两模型三维度.....	225
5.2.2	体系总览.....	227
5.2.3	oop 体系.....	229
5.2.4	klass 体系.....	231
5.2.5	handle 体系.....	234
5.2.6	oop、klass、handle 的相互转换.....	239
5.3	常量池 klass 模型 (1).....	244
5.3.1	klassKlass 实例构建总链路.....	246

5.3.2	为 klassOop 申请内存	249
5.3.3	klassOop 内存清零	253
5.3.4	初始化 mark	253
5.3.5	初始化 klassOop._metadata	258
5.3.6	初始化 klass	259
5.3.7	自指	260
5.4	常量池 klass 模型 (2)	261
5.4.1	constantPoolKlass 模型构建	261
5.4.2	constantPoolOop 与 klass	264
5.4.3	klassKlass 终结符	267
5.5	常量池解析	267
5.5.1	constantPoolOop 域初始化	268
5.5.2	初始化 tag	269
5.5.3	解析常量池元素	271
5.6	本章总结	279
第 6 章	类变量解析	280
6.1	类变量解析	281
6.2	偏移量	285
6.2.1	静态变量偏移量	285
6.2.2	非静态变量偏移量	287
6.2.3	Java 字段内存分配总结	312
6.3	从源码看字段继承	319
6.3.1	字段重排与补白	319
6.3.2	private 字段可被继承吗	325
6.3.3	使用 HSDB 验证字段分配与继承	329
6.3.4	引用类型变量内存分配	338
6.4	本章总结	342
第 7 章	Java 栈帧	344
7.1	entry_point 例程生成	345
7.2	局部变量表创建	352
7.2.1	constMethod 的内存布局	352

7.2.2	局部变量表空间计算	356
7.2.3	初始化局部变量区	359
7.3	堆栈与栈帧	368
7.3.1	栈帧是什么	368
7.3.2	硬件对堆栈的支持	387
7.3.3	栈帧开辟与回收	390
7.3.4	堆栈大小与多线程	391
7.4	JVM 的栈帧	396
7.4.1	JVM 栈帧与大小确定	396
7.4.2	栈帧创建	399
7.4.3	局部变量表	421
7.5	栈帧深度与 slot 复用	433
7.6	最大操作数栈与操作栈复用	436
7.7	本章总结	439
第 8 章	类方法解析	440
8.1	方法签名解析与校验	445
8.2	方法属性解析	447
8.2.1	code 属性解析	447
8.2.2	LVT&LVTT	449
8.3	创建 methodOop	455
8.4	Java 方法属性复制	459
8.5	<clinit>与<init>	461
8.6	查看运行时字节码指令	482
8.7	vtable	489
8.7.1	多态	489
8.7.2	C++中的多态与 vtable	491
8.7.3	Java 中的多态实现机制	493
8.7.4	vtable 与 invokevirtual 指令	500
8.7.5	HSDB 查看运行时 vtable	502
8.7.6	miranda 方法	505
8.7.7	vtable 特点总结	508

8.7.8	vtable 机制逻辑验证	509
8.8	本章总结	511
第 9 章	执行引擎	513
9.1	执行引擎概述	514
9.2	取指	516
9.2.1	指令长度	519
9.2.2	JVM 的两级取指机制	527
9.2.3	取指指令放在哪	532
9.2.4	程序计数器在哪里	534
9.3	译码	535
9.3.1	模板表	535
9.3.2	汇编器	540
9.3.3	汇编	549
9.4	栈顶缓存	558
9.5	栈式指令集	565
9.6	操作数栈在哪里	576
9.7	栈帧重叠	581
9.8	entry_point 例程机器指令	586
9.9	执行引擎实战	588
9.9.1	一个简单的例子	588
9.9.2	字节码运行过程分析	590
9.10	字节码指令实现	597
9.10.1	iconst_3	598
9.10.2	istore_0	599
9.10.3	iadd	600
9.11	本章总结	601
第 10 章	类的生命周期	602
10.1	类的生命周期概述	602
10.2	类加载	605
10.2.1	类加载——镜像类与静态字段	611
10.2.2	Java 主类加载机制	617

10.2.3	类加载器的加载机制	622
10.2.4	反射加载机制	623
10.2.5	import 与 new 指令	624
10.3	类的初始化	625
10.4	类加载器	628
10.4.1	类加载器的定义	628
10.4.2	系统类加载器与扩展类加载器创建	634
10.4.3	双亲委派机制与破坏	636
10.4.4	预加载	638
10.4.5	引导类加载	640
10.4.6	加载、链接与延迟加载	641
10.4.7	父加载器	645
10.4.8	加载器与类型转换	648
10.5	类实例分配	649
10.5.1	栈上分配与逃逸分析	652
10.5.2	TLAB	655
10.5.3	指针碰撞与 eden 区分配	657
10.5.4	清零	658
10.5.5	偏向锁	658
10.5.6	压栈与取指	659
10.6	本章总结	661

第 1 章

Java 虚拟机概述

本章摘要

- ◎ Java 语言产生的历史背景
- ◎ 编程语言跨平台的实现
- ◎ 中间语言的实现

1.1 从机器语言到 Java——詹爷，你好

当年——在 60 多年前，程序员是这样干活的：

写一段程序，将其打在纸带或卡片上，1 打孔，0 不打孔，然后将纸带或卡片输入计算机。那时候的程序都是只用 0 和 1 写成，注意，是只用 0 和 1 哟！

道理大家都懂，计算机嘛，只识别 0 和 1。

当年，程序员用于编程的 IDE 就是剪刀+胶水，只这两板斧，就能闯天下。

图 1.1 所示就是传说中的穿孔卡带，这就是当年程序员们开发出来的程序。

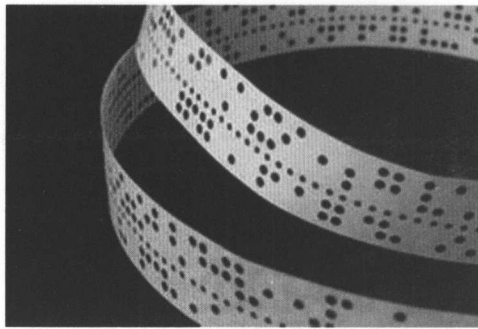


图 1.1 保存程序的卡带

请聚精会神盯着这个纸带上面的孔看 3 秒钟，计时开始：1，2，3。OK，时间到，怎么样？晕了吧！这哪里是程序，分明是个千疮百孔的纸带嘛！

什么，没晕!!! 好，那我们一起来玩个游戏吧，这个游戏就是：大家一起来找茬（相信很多经历并见证诺基亚这个手机巨人倒下以及安卓兴起这段历史的朋友们，对这款游戏一定有很深的印象）。好的，游戏开始，先看下面第一幅图（没错，下面就是使用数字 0 和 1 所绘制而成的特殊的“图画”）：

```
1010000100000000100000000
```

```
0000000110000010100000010
```

再看第二幅图：

```
1010000100000000100000000
```

```
0000000110000100100000010
```

现在，请比较这两幅图，找出其中的差异。相信眼尖的你一定很快就发现了其中的不同。其实，这段机器代码是在执行一个小学一年级的简单数学题目：1+2。

什么，还没晕！

你这分明是逼我放大招的节奏啊！那好，我就满足你一下，现在，我们稍微提升一下游戏的难度，还是两幅图，第一幅图如下：

```
1010000100000000100000000
```

```
010010110001110100000000
```

```
010010110000110100000100
```

```
0000000111010100
```



```
000000110001110100000001
```

```
1110001011110110
```

第二幅图如下：

```
101000010000000100000000
```

```
010010110001110100000000
```

```
010010110000110100000100
```

```
0000000111010100
```

```
000000110000110100000001
```

```
1110001011110110
```

现在请找出这两幅图中的差异点。

现在真的晕了吧！还是让我悄悄地告诉你差异点在哪里吧，倒数第 2 行左边开始第 12 个位置的 1 变成了 0 哦，别问我是怎么知道的，我是不会告诉你的。

其实，这段程序是在计算一个小学 2 年级的数学题目：求 1~8 的和。这段机器码中使用了循环，如果写成 Java 代码，类似这样：

```
int sum=0;
for(int i=0; i < 8; i++){
    sum += i;
}
```

游戏玩完了，怎么样？一定很无聊吧。当年的程序员们也是这么想的。设想一下，如果你恰好不幸出生在那个年代，并且恰好不幸当了一名程序员，某一天你编写了一个程序，这个程序不大，只有 2000 行，结果你恰好一不小心把其中某一行的某个 0 写成了 1，或者把某个 1 写成了 0，或者少写了一个 0，那是一件多么不幸的事！代码写错了。可这对于现代程序员而言，那都不是事，大不了断点调试一把嘛！但在那个 IDE 简陋到只有剪刀+胶水的年代，那可真是抱黄连敲门——苦到家了。你只能把眼睛瞪得跟铜锣一样大，对着这 2000 行代码，仔仔细细、一行一行地排查问题究竟出在哪里。等你“望眼欲穿”，终于发现问题后，你得重新操起家伙（剪刀和胶水），重新制作纸带，然后重新运行程序。可是，怀着万分期待的你，绝望地发现程序还是运行出错了！恭喜你，还得麻烦你老再次逐行排查这 2000 行机器码，保证不把你的眼睛看瞎。保重，不谢！

于是，大家纷纷要求改变这种反人类的工作，而改变的思路就是：既然机器码这么难以阅读、理解和排错，那就用助记符吧。于是人们开始用助记符来编写程序，编写完了，先用人脑来执行一遍（厉害吧），确保没有问题后，再将助记符手工转换成机器码，制作成孔带。

虽然那是一个不幸的年代，二战还没完全结束，全世界很多地方还有战争、饥饿和疾病，但那也是一个乱世出英雄的年代，时事造人。Grace Hopper 先生就是其中的一位牛人，他觉得这样仍然很麻烦，就开发了 A-0 system，其能够自动将助记符转换成机器码。

当时的助记符，就是所谓的汇编。

后来，人们纷纷在主流硬件平台上基于汇编进行程序开发。汇编这东西，小巧精悍，一看就懂，容易理解，上手还快，节能环保，无公害无辐射，不会引起雾霾，用过的人都说好，大家一直用它。

后来，再后来呢？再后来的事大家都知道了，聪明的前辈们先后开发出了 Fortran、B、C、C++、Delphi、VB、PHP 等各种高级语言。高级语言的出现，再一次解放了程序员的生产力，原本用汇编需要写 20 行的程序，高级语言一行代码就搞定，极大地提高了生产效率。同时，人们还开发出配套的 IDE（Integrated Development Environment，集成开发环境），使得调试程序和试错成为易如反掌的事，编程逐渐变得简单、有趣，从苦力活一跃成为一门艺术活。并且，高级语言的出现，为人类编写大型程序提供了坚实的基础，如果没有这些高级语言，很难想象现在会不会出现操作系统，以及基于操作系统的各种游戏、网络、电商等应用程序。

对于这段历史，如果真要去逐一追根溯源，写成的书用“汗牛充栋”来形容一点都不过分，不过所幸早有聪明的前辈把这段历史总结成了一句话：机器生汇编，汇编生 B，B 生 C，C 生万物。

这句话看着很熟悉有没有？一生二，二生三，三生万物，嗯，记起来了？你是不是发现了什么？

时间一晃四五十年过去了，我们穿越到了 20 世纪 90 年代初。那一年，整个程序界依然呈现百家争鸣、百花齐放的一派欣欣向荣的景象，各种编程语言各霸一方。那一年，地球东半球的人们普遍处于“通信基本靠吼，取暖基本靠抖”的生活水平（哈哈，有点夸张，不过那时候连手电筒都算家用电器，更别提空调和电话啦），而西半球的人们已经开始大量使用电视机、电话、闹钟、烤面包机等现代化生活电器。这些家用电器五花八门，由不同的公司制造和生产。由于大家技术各不相同，因此使用的电子芯片也不相同，大家需要针对不同的硬件进行程序开发。那一年，有个年轻的博士具有非常前瞻的商业眼光，看准了家电智能化的发展方向，便暗自下定决心，一定要运用自己所学，来实现家电智能化，开发出一种能够运行于各种不同硬件平台的编程语言，这样大家就不需要关注不同硬件平台的细节差异，只需将精力完全用在应用程序开发上面，这就能再次解放生产力，促进社会飞速发展。

这个人，就是詹爷，江湖人称“Java 之父”——詹姆斯·高斯林。

可是，要开发出这样一种能够横跨各种异构平台的编程语言，谈何容易？好比两个语言不

通的原始部落的人半途相遇，愣是急得大眼瞪小眼，王八盯绿豆，却一句话也听不懂。解决办法很多，但是关键要有第一个敢于“吃螃蟹”的人。詹爷，主意定了么？要不试试？

试试就试试，谁怕谁！

于是，在一个月不黑、风不高的夜晚，詹爷操起了那把刀，不，是打开了电脑，开启了一段创造不朽传奇的辉煌之路！

如今，20 多年过去了，伊人仍在，而江湖，早已不是那个江湖。当初詹爷一心想捣鼓出个“write once, run anywhere”的编程语言来一统江湖，可是江湖局势突变，家居智能化的发展并不明朗，反而是互联网异军突起。然而，Java 语言却阴差阳错地在互联网领域大展身手，乘着“互联网”这朵青云扶摇直上。到如今，Java 语言已经稳坐互联网开发第一宝座很多年。近几年兴起的大数据、分布式开发，很多中间件框架和平台也都直接基于 Java 开发，Java 语言在新的领域日渐焕发出绚烂夺目的光彩，这恐怕连当年詹爷自己都没想到。

真是世事无常！

然而，戏剧性的事情还不仅于此。在詹爷开发出 Java 语言的 20 年之后，一群志同道合者捣鼓出了一个手机操作系统——Android，其使用 Dalvik 虚拟机（虽然现在虚拟机被谷歌公司升级了，不再叫 Dalvik 了），但是语法规则完全遵循 Java 语言。经过谷歌公司的大力推广，Android 如今占据手机市场的半壁江山，并且当下随着手机与智能家居的融合愈加深化，从某种意义上说，这反而使得詹爷当年的理想得以实现。转了一大圈，终于又转了回来。此路不可谓不曲折。

詹爷功力深厚，使用独家武功秘籍打败江湖无敌手，坐拥半壁江山（Java 所占据的应用领域一直位列前茅）。我等后辈小子无才，无天赋，虽不想闻达于诸侯，却爱好追踪觅仙、寻根溯源，解密詹爷当年雄霸武林的成名功夫，其乐无穷也。

备注

（1）第一个找茬游戏中，两行机器指令对应下面这段汇编（基于 x86 平台）：

```
mov 0x1, %eax
add 0x2, %eax
```

（2）第二个找茬游戏中，那段机器指令对应下面这段汇编（基于 x86 平台）：

```
mov 1, %eax
mov 0, %ebx
mov 8, %ecx
s:
    add %ebx, %eax
    add 1, %ebx
loop s
```

1.2 兼容的选择：一场生产力的革命

詹爷想一统江湖，希望开发出一款编程语言，能够兼容所有的硬件平台和操作系统。但是怎样实现呢？这是一个问题。

在当时，要实现兼容性，并不难。很多编程语言都具备这种能力，例如 C 语言。问题的关键在于，怎样才能在开发层面实现真正的平台无关性？

我们先看看 C 语言是如何实现兼容性的。C 语言实现系统兼容性的思路很简单，那就是通过在不同的硬件平台和操作系统上开发各自特定的编译器，从而将相同的 C 语言源代码翻译为底层平台相关的硬件指令。虽然这种思路很棒，但是仍然有明显的缺点，当涉及系统调用时，开发者仍然要关注具体底层系统的 API。例如，在 C 语言中创建一个线程，如果你是在 Linux 平台上开发，那么 C 语言程序必须要这样写：

清单：示例程序

作用：在 Linux 平台上使用 C 语言创建线程

```
#include <stdio.h>
#include <pthread.h>
//子线程函数
void threadCallBack(void)
{
    int i;
    printf("This is a pthread.\n");
}

int main(void)
{
    pthread_t id;
    int i,ret;
    ret=pthread_create(&id, NULL, (void *)threadCallBack, NULL);
    if(ret!=0){
        printf ("Create pthread error!\n");
        exit (1);
    }

    printf("This is the main process.\n");
    pthread_join(id,NULL);
    return (0);
}
```

如果你是在 Windows 平台上开发，那么 C 程序必须要这么写：

清单：示例程序

作用：在 Windows 平台上使用 C 语言创建线程

```
#include <stdio.h>
#include <windows.h>
//子线程函数
DWORD WINAPI ThreadFun(LPVOID pM)
{
    printf("This is a pthread\n");
    return 0;
}

int main()
{
    printf("This is the main process.\n");
    HANDLE handle = CreateThread(NULL, 0, ThreadFun, NULL, 0, NULL);
    WaitForSingleObject(handle, INFINITE);
    return 0;
}
```

可以看到，在 Linux 平台上，开发者需要知道 Linux 平台所提供的创建线程的接口是 `pthread_create()`；而在 Windows 平台上，开发者需要知道 Windows 平台所提供的创建线程的接口是 `CreateThread()`。另外，在 Linux 和 Windows 平台上，C 程序需要引用不同的头文件，并且所调用的创建线程的两种 API 的人参和返回值也不相同。

这就是本节一开始所说的，开发层面的平台相关性。即，开发者需要在开发时就考虑和了解各种平台的差异性。

由此可见，如果开发者想要使用 C 语言来开发一款既兼容 Linux 平台又兼容 Windows 平台的程序，不仅需要熟悉 C 语言本身的 API，也要熟练掌握 Linux 和 Windows 上的相关 API。这带来的直接后果就是，开发者为了开发一个特定平台上的功能，不得不先花费巨大的精力去熟悉该平台的特性和 API。

可能很多人对这种额外付出的成本没有概念，这里通过一组数据来对比一下，使大家对此有一种感性认识：

平台/编程语言	API 数量
C 编程语言	200 多个
Linux	300 多个
Windows	2000 多个

我们拿 C 语言举例，200 多个 API，要达到入门级水平，对于聪明的你，一定很快，你可以在分分钟内编写出第一个 demo，并在屏幕上成功打印出一行“Hello world!”。但是要精通，

事情就不简单了。你不仅要熟知这些 API 的一般用法，还需要通过大量实践，熟知里面可能存在的各种坑。更需要通过参与实际的商业项目开发，切实体会怎样才能安全、高效地使用这些 API，使你的商业程序既能拥有高性能，又能一直稳定可靠地运行下去。如果你是一个像詹爷那样拥有远大理想和崇高追求的程序员，那么你一定会追求做一个合格的架构师，你一定会基于 C 语言的 API 另外封装出自己的一套 API，使你的程序拥有良好的灵活性和扩展性。除了 C 语言的 API，还得另外熟悉 C 语言的各种语法、编译规则，利用这些规则你才能构建出一种合理的软件架构。

这么一整套东西整下来，再聪明的人，最快也需要三四年的时间，才能达到高级程序员的水平。由此可见，要想精通某一个平台或者编程语言，是一件很费时费力的事，并没有速成法，江湖上所谓的“一周精通xxx”的宣传语，我们一看就知道是假的。

詹爷当年也是这么想的。人的一生是有限的，我们应该把有限的生命，投入到无限的应用开发上去，而不是浪费在无限的、层出不穷的底层细节上。

詹爷觉得开车只要学会怎样启动，怎样刹车，怎样踩油门和离合器就可以了，没必要让驾驶员关注是啥发动机，不管是涡轮增压还是自然吸气，开车的人都不需要关注，更不需要知道怎样才能操控好，那是专业车手的事。

詹爷认为，简单的，才是最美的。一个应用开发者，只需要关注功能实现本身，不应该去关注底层的细节，这样才能将生产效率提上去。

否定别人总是一件很容易的事，但要成就自己，却很难。既然通过编译器来实现兼容性，是如此地低效和费电，那应该怎样做，才能既不费电，又能实现兼容性呢？

这真是一个让人头疼的问题！

既然铁了心要解决这个问题，那就让我们再次好好分析分析问题，看看问题的焦点在哪里。现在，让我们看看所面临的问题，一共有两个问题：

- ◎ 现有的技术能够实现兼容性，但是成本太高，效率太低，太费电。
- ◎ 无法在开发层面做到平台无关性。

我们所要达成的目标也有两个：

- ◎ 实现兼容性。
- ◎ 开发者不需要关注底层平台的异构性，就能实现兼容性。

请注意我们的目标，目前两个目标里，其实兼容性已经实现了，问题的焦点就在于第 2 点，我们要做到让开发者对底层细节差异无感，能够写出可以兼容所有底层平台的程序。

还是举上面创建线程的例子。假设有这么一种编程语言，开发者编写了一条指令，说我要

创建一个线程。当编写好的程序运行在 Linux 平台上时，这条指令就自动被转换成调用 `pthread_create()` 接口；当程序运行在 Windows 平台上时，这条指令就自动被转换成调用 `CreateThread()` 接口。这样开发者就不需要关注不同底层平台上的 API 了，只需要知道有这么一条指令就可以了。

于是，中间语言（IL）就产生了。

虽然我从来没有和詹爷面对面交谈过，但我想，当年，詹爷也一定这么想过。而事实上，詹爷就是这么干的。詹爷定义了字节码规范，字节码就是中间语言指令。同时，詹爷开发了虚拟机，由虚拟机负责将字节码转换成不同平台上的特定 API 调用。由此，我们的目标终于得以达成，开发者终于不需要关注底层硬件和操作系统层面的细节，一切都由虚拟机解决，开发者只需要熟知 Java 语言规范和 API 即可实现特定功能开发，这极大地提高了生产效率，也大大降低了编程的难度和门槛，为如今全世界几百万 Java 开发者提供了饭碗。就冲这一点，我们亲切地称呼詹姆斯·高斯林为“爷”，我想大家应该都能接受吧！

OK，上面我们简单地回顾了一下当时 Java 语言产生的背景、面临的问题以及解决办法，我们知道，Java 语言从一开始就与其他语言的定位不同。现在我们知道，一款编程语言要实现兼容性，至少有两种办法：一种是通过编译器实现兼容，一种是通过中间语言实现兼容。关于这两者的区别，詹爷自己用 4 个单词做了精辟的总结，那就是：

`write once, run anywhere!`

这后一句，`run anywhere`，自然就是指兼容性，无论是编译器还是中间语言，都能实现 `run anywhere`。区别就在前一句：`write once`。使用编译器实现兼容性时，不能实现 `write once`。例如上文所举的使用 C 语言创建线程的例子，如果你的程序一开始是为 Linux 编写的，那么当你想把程序迁移到 Windows 平台上时，你必须得修改程序，把里面涉及线程创建的代码全部改成 Windows 的接口。当你的程序很大，涉及大量的系统调用时，程序中必将有很多地方都需要改写，这种改写的工作量不可谓不大，伴随而来的是高风险。你不仅要精通 Linux API，也要精通 Windows API，否则很容易一不小心中招、踩坑，为程序埋下隐患。

而使用中间语言，就用不着考虑这些想想都令人头大的问题啦！你只需要写好程序，实现程序的逻辑。程序写好后，无论你想部署到 Linux 平台上，还是想部署到 Windows 平台上，都不用修改哪怕一句代码！所有兼容性的工作都由虚拟机帮你做好了。

这种生产效率的提高绝对是革命性的！从此，Java 语言虽然没有直接走上一条康庄大道，但却在很多领域都大放异彩。“条条道路通罗马”，Java 的出现，大大缩短了商业软件的开发周期，极大地提高了 IT 业的生产效率，极大地解放了程序员的创造力。

在本节末尾，让我们一起来做个总结。

一款编程语言兼容底层系统的方式大抵上分为两种。

1. 通过编译器实现兼容

例如 C、C++ 等编程语言,既能运行于 Linux 操作系统,也能运行于 Windows 操作系统;既能运行于 x86 平台,也能运行于 AMD 平台。这种能力并不是编程语言本身所具备的,而是由编译器所赋予。针对不同的硬件平台和操作系统,开发特定的编译器,编译器能够将同样一段 C/C++ 程序翻译成与目标平台匹配的机器指令,从而实现编程语言的兼容性。

但是通过编译器实现兼容性时,如果涉及系统调用,往往都需要修改程序,调用特定系统的特定 API,否则程序迁移到新的平台上之后,无法运行。

2. 通过中间语言实现兼容

Java、C# 等语言,都属于这种兼容方式。

Java/C# 程序被编译后,生成中间语言 (ML),中间语言指令由虚拟机负责解释和运行。虚拟机在运行期将中间语言实时翻译成与特定底层平台匹配的机器指令并运行。无论程序最终运行在何种底层平台上,源代码被编译生成的中间语言指令都是相同的,中间语言的兼容性由虚拟机负责完成。

通过编译器实现兼容性,由于源代码被直接编译成了本地机器指令,因此其执行效率非常高。而这正是中间语言的软肋。Java 语言刚问世那几年,就一直因为其性能低下而被嗤之以鼻。但是随着 Java 语言版本的不断更新,随着大家对改善其性能所做出的持之以恒的努力,如今 Java 性能已经相当高,甚至比 C/C++ 程序性能还要高。这是因为 Java 虚拟机内部对寄存器进行了大量手工优化,在某些场景下,人工优化自然会比 C/C++ 编译器所做的机器优化效果要好很多。

1.3 中间语言翻译

上一节,我们讨论了既能实现兼容性,又能自动处理底层系统调用的又快又好的办法,那就是使用中间语言。可是 CPU 是不认中间语言的,它无法直接执行中间语言。为了使中间语言能够被 CPU 执行,虚拟机必须将其翻译成对应机器上的机器指令。

于是接下来的一个课题就是,怎样将中间语言翻译成对应的机器指令并得以执行。注意,虽然这是一个课题,但却包含两个问题哟!一个问题是怎样把中间语言翻译成对应的机器指令,另一个问题是翻译完了还要能够执行。

这里先进行总述。将中间语言翻译成对应的本地机器指令,可以使用 C 语言为每一个 Java

字节码指令写一个对应的实现函数，也可以直接为中间语言生成对应的本地机器码并通过 JMP 方式跳转到机器码来执行字节码指令。

下面我们针对这两个问题分别讲述。

1.3.1 从中间语言翻译到机器码

还记得本文开篇中所提到的一句话——“机器生汇编，汇编生 B，B 生 C，C 生万物”吗？

虽然现代的程序员们，手里拿着 Java、C# 等这些高端武器，沐浴在高级语言所带来的满满的幸福感中，分分钟就能编写出一个强大的程序，不需要拥有太多专业知识，仅仅依靠先进的武器，就能发出强大的威力，制敌获胜。

可是，这些高级语言的背后，却是虚拟机在辛辛苦苦、勤勤恳恳地干活，实现“万物到 C，C 到汇编，汇编到机器”的逆转换，这里的“万物”，自然也包括 Java 的中间语言——字节码指令。

将字节码指令翻译到对应的机器码，一种可行的办法是，使用 C 程序，将字节码的每一条指令，都逐行逐行地解释成 C 程序。当执行字节码的程序——JVM（Java 虚拟机）程序本身被编译后，字节码指令所对应的 C 程序被一起编译成本地机器码，于是虚拟机在解释字节码指令时，自然就会执行对应的 C 程序所对应的本地机器码。

使用语言解释起来，懂的人自然一看就明白，但是对于缺乏相关专业基础的同学，未必一下子就能看明白。所以这里举个例子进行说明。

1.3.2 通过 C 程序翻译

例子很简单，计算两个正整数之和。首先，假设使用 C 程序编码，程序可以写成这样：

清单：main.c

作用：C 语言求和

```
#include <stdio.h>
int run(int a, int b);

int main(){
    int a=5;
    int b=3;
    int r = run(a,b);
    printf("r=%d\n", r);
    return 0;
```

```
}
int run(int a, int b){
    return a+b;
}
```

假设我们发明了某种中间语言，该中间语言定义了一条指令来实现两个正整数相加，这条指令的助记符是 `iadd`，其对应的数字唯一编号是 `0x01`，同时假定我们开发了一款虚拟机来解释这条指令，那么解释程序将变成如下这样：

清单：main.c

作用：虚拟解释器

```
#include <stdio.h>
int run(int a, int b, int code);

int main(){
    int a=5;
    int b=3;
    int code=0x01;//0x01
    int r = run(a,b,code);
    printf("r=%d\n", r);
    return 0;
}

int run(int a, int b, int code){
    if(code==0x01){
        return a+b;
    }
    return -1;
}
```

这段示例程序中的 `run()` 函数，可以看做是执行引擎（哈哈，这可能是世界上最精简小巧的虚拟机执行引擎啦）。执行引擎接收操作数和指令编码，判断指令编码是否是 `iadd`，如果是 `iadd` 指令，便对入栈的两个操作参数执行加法运算，并返回结果。

是不是很简单？

第一代 JVM 的执行引擎就是这么简单。

虽然通过 C 程序对中间语言进行解释，程序简单明了，逻辑清晰易懂，然而这种方式却有一个比较大的缺陷——效率低下。所以第一代 Java 虚拟机被广为诟病和吐槽，因为效率实在是太低了。对此，即使身怀绝技的詹爷也没有什么好的办法去优化，也不得不另外想辙。大象踩死蚂蚁是件轻松得不能再轻松的事，但不管你怎么努力，一只蚂蚁也绝对不可能踩死一只大象。此路不通，那就换条道。

这辙，还真有，那就是借助于老祖宗的东西，返璞归真。既然使用 C 这种高级语言还是效率低，那就直接翻译成机器码吧，这样一定可以很快了吧！

1.3.3 直接翻译为机器码

将中间语言直接翻译为机器码，办法有很多。追根到底，我们还是利用了 CPU 执行代码的原理。要让 CPU 执行一段代码，只需将 CS:IP 段寄存器指向到代码段入口处即可。各位道友注意了，本节将开始接触到汇编语言了，不过不用担心，懂汇编的自然一看就明白，不懂汇编的，多看几遍也就懂了。全书很多地方都引用了汇编指令，但是加起来也不超过 5 个，不信你可以看完全书再来统计，看我有没有骗你（哈哈）。对于聪明的你，区区 5 个汇编指令还在话下吗？

这里首先解释一下啥是 CS 与 IP。这是物理 CPU 内部的两个寄存器。对于一台物理机器而言，这两个寄存器是最重要的寄存器，因为 CPU 在取指令时便完全依靠这两个寄存器。CS 寄存器保存段地址，IP 保存偏移地址。CS 和 IP 这两个寄存器的值能够唯一确定内存中的一个地址，CPU 在执行机器指令之前，便通过这两个寄存器定位到目标内存位置，并将该位置处的机器指令取出来进行运算。函数跳转的本质其实便是修改 CS 和 IP 这两个寄存器的内容，使其指向到目标函数所在内存的首地址，这样 CPU 便能执行目标函数了。Java 虚拟机要想让物理 CPU 直接执行 Java 程序所对应的目标机器码，也得修改这两个寄存器才能实现。

修改 CS:IP 段寄存器的办法有很多，既可以使用汇编直接修改，也可以在高级语言中通过语法糖的形式修改，C 语言中就有这样的语法糖（也许叫语法规则更加恰当一些，但是某种特定的写法最终都由编译器来进行转换，到了机器码层面，已经没有这些规则了，因此将其称为语法糖也未尝不可）。

C 语言中提供了一种办法，可以将 CS:IP 段寄存器直接指向到某个入口地址，这种办法就是定义函数指针（注：不是指针型函数，这两个概念相差很大，完全不同，读者千万不要搞混）。下面提供一个示例：

清单：示例程序

作用：使用 C 程序提供的语法糖修改 CS:IP 段寄存器的指向

```
#include <stdio.h>
int run(int a, int b);

int main(){
    int a=5;
    int b=3;

    int (*fun)(int,int); // 定义函数指针
    fun=(void*)run; // 初始化函数指针，使其指向 int run(int,int) 函数入口
```

```

    int r=fun(a,b); // 执行函数
    printf("r=%d\n", r);
    return 0;
}

int run(int a, int b){
    return a+b;
}

```

在本示例中,我们先定义了一个函数指针 `fun`,接着将其指向 `int run(int,int)` 函数入口,当 `run()` 函数被操作系统加载后,其机器码指令将被保存到代码段中,`fun` 指针就指向该代码段的首地址。最后,通过 `int r=fun(a,b)` 执行 `fun`,由于 `fun` 指向 `run()` 函数入口,因此系统最终执行的实际上是 `run()` 函数。

在 Linux 上运行这段程序,最终屏幕上将正常输出“`r=8`”这行文字。

其实, `fun=(void*)run` 这句代码之所以得以正常运行,完全是因为编译器提供了这种语法支持,因此我们可以将其视为 C 语言的语法糖。这句代码被编译后,实际上相当于修改了 `CS:IP` 段寄存器的指向,因此当 CPU 运行到这里后,由于 `CS:IP` 指向到了 `run()` 函数首地址,因此程序就跳转到 `run()` 函数。

上面讲了一堆,可是这与我们本节的主旨有何干系呢?我们本节要解决的问题是,如何将中间语言直接翻译成机器码。虽然本示例与本节主旨似乎没有什么关系,但是通过本示例,我们知道了如何使用 C 语言所提供的语法糖来修改 `CS:IP` 段寄存器,有了这个法宝,我们就能实现我们的目标。因为 C 语言程序最终也会被编译成本地机器指令,因此我们可以预先定义一串本地机器指令,然后直接通过 C 语言所提供的语法糖,将 `CS:IP` 段寄存器指向这串机器指令,这样 C 程序就能直接动态执行机器码。

好了,废话少说,下面直接上菜:

清单: 示例程序

作用: 使用 C 程序提供的语法糖,让 `CS:IP` 直接指向一串机器码

```

#include <stdio.h>

/**
 * code 数组中保存的一串数据,正好是机器码指令编码,CPU 可以直接执行
 * 注:
 * (1) 对于计算机,一个数据既可以被当作指令执行,也可以被当作一个数据
 * (2) 当这个数据被 CS:IP 段寄存器指向时,就可以当作代码执行
 */
const unsigned char code[]="\x55\x89\xe5\x8b\x45\x0c\x8b\x55\x08\x01\xd0\x5d\xc3";

```

```

int main(){
    int a=5;
    int b=3;

    int (*fun)(int,int); // 定义函数指针
    fun=(void*)code; // 初始化函数指针, 将其指向 code 机器码的入口

    int r=fun(a,b);
    printf("r=%d\n", r);
    return 0;
}

```

本例所要实现的功能与上例一样, 都是实现对两个正整数求和。

本例也同样使用了 C 语言所提供的语法糖, 通过 `fun=(void*)code` 这样的方式达到间接修改 CS:IP 段寄存器的指向的目的。但是本例与上例的不同之处在于, 在上例中, `fun` 指针是直接指向了 `int run(int,int)` 函数, 而本例中, `fun` 指针却是指向了一个 `char` 数组首地址。

`char` 数组中保存了一串数字, 这些数字其实是 x86 平台上的一串机器指令, 因此 CPU 可以直接将其当做指令来执行。

这是一个特别神奇的神器! 很多大牛都用它实现了很多匪夷所思的功能。集中起来而言, 就是大家用它实现了各种动态功能, 各种在运行期动态修改程序走向的功能。有了这件神器, 要实现一个高可扩展性的架构, 再也不是难事了!

詹爷早就看好了这件神器, 并且是此中高手。在 JVM 的后续版本中, 正是这件神器, 挽救了 Java 语言, 使其性能得到突飞猛进的提高, 在某种程度上快赶上甚至超越 C 和 C++ 语言的速度了, 其助推 Java 语言大步向前进!

咦? 貌似还少了啥事?

刚才只顾夸夸其谈, 忘了本节主旨了, 我们还没有实现由中间语言直接翻译为机器码的伟大目标呢! 但是我相信, 聪明的你一定早就想到了办法。对, 果然与你想的一样, 我们既然都能在 C 语言中直接动态执行机器码了, 注意, 是动态哟! 我们只要将中间语言指令直接翻译为机器码, 然后让 CS:IP 直接指向这段机器码, 问题不就解决了吗?

是的, 事情的确就是这么简单! 现代 JVM 的确就是这么干的。

不过, 在 JVM 里面又不完全是这么干的, 但这不是很重要, 重要的是, 我们已经有办法、有能力完成将中间语言直接翻译成机器码并动态执行的宏伟目标, 这是一个具有战略意义的里程碑, 决定生死存亡!

注: 本示例中, `code` 数组中所保存的一串数据, 其实对应下面一组机器指令:

```

55          push    %ebp
89 e5      mov     %esp,%ebp
8b 45 0c    mov     0xc(%ebp),%eax
8b 55 08    mov     0x8(%ebp),%edx
01 d0      add     %edx,%eax
5d         pop     %ebp
c3         ret

```

1.3.4 本地编译

虽然将中间语言直接翻译为机器码并直接运行，其效率相比使用 C 语言来解释执行，已经提高了很多，但是，由于中间语言有自己的一套内存管理和代码执行方式，因此，实现同样的功能，虽然使用中间语言只需写几行代码，但是翻译后的机器码，比直接编写机器码，还要多出很多指令。指令数量增多，意味着在同样的硬件平台上，执行时间成本必然增加，因此其运行效率仍然不够高。

即使与同样属于高级语言的 C 语言相比，它们实现相同的功能，C 语言编译后所生成的机器码，也比中间语言直接翻译成的机器码，在数量上要精简很多。

例如，使用 C 语言对两个正整数求和。示例程序如下：

清单：示例程序

作用：使用 C 语言求和

```

int add(int a, int b){
    return a+b;
}

```

本地编译该程序，得到的机器码如下（使用汇编助记）：

清单：示例程序

作用：C 语言求和程序编译后的机器码

```

pushl    %ebp
movl%esp, %ebp
movl12(%ebp), %eax
movl8(%ebp), %edx
addl%edx, %eax
popl%ebp
ret

```

可以看到，add()函数总共仅包含 7 条机器指令，即可完成求和运算，如果去掉 pushl %ebp 等入栈、出栈的辅助性指令，只需要下面 3 条机器指令即可完成：

清单：示例程序

作用：C 语言求和程序编译后的机器码

```
movl12(%ebp), %eax
movl8(%ebp), %edx
addl%edx, %eax
```

而如果使用中间语言来执行求和运算，翻译后得到的机器码指令将会多出几个数量级。这里以 Java 为例。首先，编写 Java 源程序：

清单：示例程序

作用：Java 语言求和程序

```
class A{
    public int add(int a, int b){
        return a+b;
    }
}
```

这段 Java 程序编译后生成的字节码如下：

```
0: iload_1
1: iload_2
2: iadd
3: ireturn
```

而每一条字节码指令最终都会对应一大堆机器指令，机器指令的数量远超 C 语言编译后的机器指令数量。

由此可见，中间语言由于其本身不能被 CPU 执行，为了能够被 CPU 执行，中间语言在完成同样一个功能时，需要准备更多便于自我管理的上下文环境，最后才能执行目标机器指令。准备上下文环境最终也是依靠机器码去实现，因此中间语言最终便生成了更多机器码，当然执行效率就降低了。

为了能够进一步提升性能，JVM 提供了一种机制，能够将中间语言（字节码）直接编译为本地机器指令。

可能聪明的你马上会想到这样一个问题，既然中间语言在运行期能够被逐个直接翻译成机器码，那么在编译期不也能吗？例如对于 Java 源代码，可以先在本地编译成字节码，再将字节码指令逐个替换为机器指令，这样最终不就生成了可直接被 CPU 执行的、由机器码组成的程序了吗？

这个思路的确可以一试，例如安卓和部分 JVM 所实现的 AOT（ahead of time）特性便是这方面的尝试，但是这种方式并没有减少机器指令的数量级问题。

事实上，JVM 的大牛们在 JIT（即时编译）、内存分配等方面倾注了大量心血，想出了很多

天马行空而又切实可行的好主意，能够对热点代码进行大幅度指令优化，将 Java 程序的执行效率大幅提升。正是由于 JVM 可以在运行期基于上下文链路进行各种优化，因此优化后的指令质量比 C/C++ 编译出的指令质量更高，因此才会有部分 Java 程序性能反而超过 C/C++ 程序的现象。如果离开了这些动态优化，Java 程序的执行效率是无论如何也提不上去的。

1.4 神奇的指令

Java 语言想要一统江湖，兼容各种平台，又要实现“write once, run anywhere”的伟大梦想，只有依靠“中间语言”这一条通道了。思路是有了，但是具体如何执行，或者说如何才能实现呢？中间语言究竟长啥样，谁也不知道。

前面讲过，在使用 C 语言或汇编或 C++ 进行底层开发时，必须熟悉硬件平台本身所提供的指令，或者熟悉底层软件平台所提供的 API。在不同的平台上实现同一种功能，需要调用不同的底层接口或指令。同时，编译器也直接依赖于平台，大家需要为不同的平台开发不同的编译器。

詹爷看着这样一种现状，口中轻轻说出一个字：NO！

詹爷觉得这种方式简直太 low 了，他觉得一款好的编程语言应该是这样的，不管程序员在何种软硬件平台上开发，如果要想实现同样一个功能，他只需要调用同一个接口，他不需要感知这个平台与其他平台之间有什么差异。至于把统一接口翻译成对应的机器的指令这事，就交给虚拟机来做吧，程序员不要再关注这事了。接口统一了，就相当于两个不同的机器说的是同一种语言，这样来实现跨平台岂不是轻而易举？如果詹爷是上帝，要他出访世界各国，他肯定不愿意学习各个国家的语言，他一定会带着精通世界各国语言的随从，他只讲国际语言，具体翻译的工作，就交给随从干吧。这个随从就类似于 Java 虚拟机。

从“统一接口”这个角度看，中间语言应该是你知、我知、所有的开发者都知道，但是唯独底层的物理 CPU “不知道”的一种语言。这样看来，Java 语言本身岂不就符合“中间语言”的标准？可是别忘了，中间语言还得有一个必须什么都懂的人，这个“人”就是 Java 虚拟机啦。虚拟机就是全世界广大程序员免费而又无比专业的贴身翻译，它必须负责将这种“中间语言”精确地翻译成对应机器平台的机器指令。可是很遗憾，虚拟机这位“贴身翻译”读不懂 Java 程序。伤心啊！

所以，Java 语言不是中间语言。

为啥虚拟机读不懂 Java 程序呢？这还得从语言的人性化谈起。高级编程语言之所以高级，就在于其语法和表达规范遵循人类的思维习惯，但是这不符合机器的思维，即使虚拟机也不行。

尤其是 Java 语言，由于比其他语言更加“面向对象”，其字面含义带有更加彻底的人类主观色彩，但是机器就不能理解了，机器只认得内存、堆栈，其他一概不认，所以 Java 虚拟机读不懂 Java 语言倒也情有可原。不过也并非所有的虚拟机都不懂得面向对象的语言，JavaScript 执行引擎就是个例外——JS 脚本不需要编译就能被 JS 引擎直接执行，虽然这么玩也行，但是 JS 的执行引擎不跨平台啊，这与 Java 语言的远大理想相去甚远，道不同不相为谋。

既然 Java 语言本身不能作为中间语言，可是中间语言又是理想的实现跨平台的技术方向，这可怎么办呢？这个问题让我们的大脑很费电啊！不过詹爷觉得 so easy，解决办法很简单，为虚拟机这位“贴身翻译”再配一个翻译，负责将 Java 程序翻译成理想中的“中间语言”不就行了吗？好，那就这么定了，咱就开发个编译器吧，通过编译器将 Java 语言翻译成中间语言，然后再交给虚拟机，其再将中间语言翻译成对应机器平台上的指令。

解决了中间语言怎么实现的问题之后，詹爷要做的一件重要的事就是，决定中间语言长啥样。最终的结果大家都知道了，这个所谓的中间语言就是 Java 字节码指令集，没错，这就是中间语言。虽然 Java 语言是面向对象的，符合人类思维习惯的，但是 Java 指令却是刻板的，不知啥是对象，只知道压栈，读写局部变量表，调用目标方法，等等。

詹爷定义的通用指令集如下：

```
iconst_0, 将自然数 0 压入操作数栈。
iconst_1, 将自然数 1 压入操作数栈。
iload_0, 将索引为 0 的局部变量表的 int 型数据压入操作数栈。
istore_1, 将操作数栈栈顶的 int 型数据写入索引为 1 的局部变量表中。
// ...
```

在 JVM 源代码中，定义了 Java 语言的全部指令集，如果你平时稍微接触过字节码，那么你对上面贴出来的这部分指令一定不陌生，例如，iconst_0、istore_1，等等。

很多人一定还知道，Java 的所有指令都使用 8 位二进制描述，因此，Java 的指令总数不超过 255 个。对上面这套指令集，你看不懂也没关系，但是多少瞄上一眼，并没有任何坏处，好歹先混个脸熟，艺多不压身。

有了这套通用的指令集，一统江湖就有希望了。不过这看似简单，实则不简单。表面看来，一共才区区 200 多个指令，但这正是作者深厚功力的体现。大家都知道，指令集一般是计算机硬件才有的东西，而作者却在软件层面定义了一套同样的东西。但是，软件本身不具备执行程序的能力，软件最终还得依靠硬件指令才能完成逻辑计算。因此，一套好的软件指令必须不能超出硬件指令所能表达的计算能力，同时又要对硬件指令进行高度抽象与概括。换言之，如果你定义了一套与硬件指令集完全一模一样的软件指令集，那大家还用你干嘛呀，不如直接用硬件指令得了。

所以，詹爷所设计的指令集必定有其独到之处。那么独特在哪里呢？不急，本书后续很多内容都会涉及 Java 指令集，有的是时间，慢慢品味。在品味之前，我们得先普及一下必备的专业知识。如同古玩收藏家，就算给你一幅《滕王阁序》的真迹，但是如果你不识货，你也会当成赝品，糟蹋了天物。古玩家玩金，玩玉，玩石，玩各种器具，靠的是眼观、耳听、鼻闻、舌舔，咱们玩詹爷的指令集，不用那么讲究，只要用心即可，并且你还不用担心假冒伪劣问题，詹爷的指令集绝对货真价实，绝对正品，假一赔十。

1.4.1 常见汇编指令

由于本书主要讲解 Java 虚拟机执行引擎的内部实现机制，而 Java 虚拟机的执行引擎有太多地方直接使用了机器码实现，因此本书很多地方都不可避免地会接触到汇编指令。不过汇编指令并没有想象中的那么可怕，至少不太聪明的我在看了几十遍之后也能看懂不是（哈哈）？下面简单介绍 5 个常见的汇编指令，也就 5 个，熟悉了这几个指令，虽说不能让你在 JVM 的世界里“横着走”，但也能让你在 JVM 的世界里“行走江湖、独步天下”了。

大部分机器指令集都支持以下 5 类计算。

1. 数据传送指令

这些指令主要在寄存器与内存、寄存器与输入/输出端口之间传送数据，例如：

```
// 将自然数 1 传送到 eax 寄存器
```

```
movl 1, %eax
```

```
//将栈顶数据弹出至 eax 寄存器
```

```
pop %eax
```

注：由于机器指令是二进制，写出来谁也看不懂，因此这里使用汇编指令代替，下面也全部这样表达。汇编本来就是机器指令的助记符。

可以这么说，数据传送指令几乎是任何硬件系统都必须支持的指令。

2. 算术运算指令

包括算术基本四则运算、浮点运算、数学运算（正弦、反弦等）等。例如：

```
// 将自然数 3 与 eax 寄存器中的数累加，并将结果存储进 eax 中
```

```
add 3, %eax
```

```
// 对 ebx 寄存器中的数增 1
```

```
inc %ebx
```

3. 逻辑运算指令

与、或、非、左移、右移等指令，都属于逻辑运算指令。例如：

```
// 将 eax 中的数左移 1 个二进制位
shl %eax, 1

// 对 al 寄存器中的数和操作数进行与操作
and al, 00111011B
```

4. 串指令

连续空间分配，连续空间取值，传送等，都要使用串指令。很多高级编程语言都支持字符串运算，如果硬件没有串指令，不敢想象计算机的世界会变成什么样。

5. 程序转移指令

If...else 判断、for 循环、while 循环、函数调用等，都需要依靠程序转移指令，否则程序无法跳转。没有这些指令，程序不能模块化，无法被分隔成一个一个方法，更无法通过循环来解决很多重要的问题。常见的程序转移指令包括 jmp 跳转、loop 循环、ret 等。

好，基本专业知识的扫盲到这里先告一段落，接下来我们就可以先简单品味一下詹爷的虚拟指令集了。作为一个有良心的作者，有必要在这里先善意提醒一下大家，在研究 JVM 源码的过程中，肯定不会绕过汇编这一道坎，所以，本书会时不时地讲点汇编知识，但不会一下子讲很多。你呢，如果看不懂，也不要担心，我会尽量使用通俗易懂的方式让你懂。如果你实在看不懂，就权当是看小说罢了，无论什么语言，只要理解了其中的道理，其实都一样一样的。当然，如果能看懂，那是最好不过的，你会感受到更多的乐趣，你会感叹，啊，原来程序是可以这么写的，原来内存是可以这么划分的，原来函数是可以这么调用的！

1.4.2 JVM 指令

詹爷的指令集比上述硬件指令集更加丰富，这是因为 Java 是面向对象的编程语言，自然要有一套支持类型操作的特殊指令。总体而言，詹爷设计的指令集分为以下几部分（可能不同的人有各自不同的一套分类标准，但笔者认为基本大同小异，看你从哪个角度来分类）。

1. 数据交换指令

对 JVM 稍有了解的人都知道，JVM 内存分为操作数栈、局部变量表、Java 堆、常量池、方法区。既然有这些内存区域，那么必须要有指令支持数据在这些内存区域之间的传送和交换。例如，当你在 Java 方法中访问一个静态变量时，那么其运算过程必然伴随 JVM 将数据从常量

池传送到操作数栈的指令调用。这与硬件指令不同。以 x86 为例，一个在硬件上直接执行的程序，其内存一般分为寄存器、数据段、堆栈、常量区、代码段，CPU 为了完成运算，必然要涉及将数据从这些内存区域传送到寄存器的指令调用。

JVM 执行逻辑运算的主战场是操作数栈（iinc 指令除外，该指令可以直接对局部变量进行运算），注意，是操作数栈哦，很重要哦！不管你把数据放在堆栈中，还是放在常量池中，你要执行运算，最终 JVM 都会将数据传送到操作数栈中。而硬件执行逻辑运算的主战场是寄存器，不管你把数据放在数据段中，还是代码段，最终 CPU 都会将数据传送到寄存器中。逻辑运算完成后，再把结果转移出去。

这段文字看着有些抽象，若是配上例子，那是极好的。既然提到了战场，就拿打仗来说事。假设主战场定在 X 地区，为了取得战争胜利，作战方需要源源不断地将军队、火药、枪支、后勤物资、医疗队等都运到 X 地区。X 地区就好比是 JVM 的操作数栈，而军队、火药、物资等，都可以认为是数据。作战司令通过一道道军令来调动军队、火药、物资等，军令就类似于计算机指令。只要作战命令一下达，整个系统立马就运作起来了，一道道军令将军队、火药、物资等“数据”从各个地区“传送”到了战场上，然后又有一道道军令，将战场上的“数据”转移出去。所以，指挥作战就像编程序，或者反过来，你把编程想象成打仗，倒也是一件很有趣的事，枯燥为神奇。其实，历史上那些伟大的军事家最后都会总结出这样一句话：战争就是一种艺术。同样，虽然计算机只有短短几十年的历史，但无数牛人都总结出一句话：编程，就是一门艺术活儿。看来，天下牛人一般牛，无论玩什么，最后都能玩出艺术之美，这才是玩家的最高境界。

JVM 标准提供了丰富的数据交换指令，例如 iload、istore、lload、lstore、fload、fstore、dload、dstore、ldc、bipush 等指令，詹爷用这些指令来实现操作数栈和局部变量表之间的数据交换（这些在后文都会讲到）。JVM 规范还提供了像 getfield 和 putfield 这样的指令，詹爷用这些指令来实现 Java 堆中的对象的字段和操作数栈之间的数据交换。JVM 规范还提供了像 getstatic 和 putstatic 这样的指令，詹爷用这些指令来实现类中的字段和操作数栈之间的数据交换。JVM 规范还提供了像 baload、bastore、caload 和 castore 这样的指令，詹爷用这些指令来实现 JVM 堆中的数组和操作数栈之间的数据交换。

至此，聪明的你应该能够想到，JVM 并不是随随便便就将内存分成了操作数栈、局部变量表、Java 堆、常量池、方法区这几个区域的，人家都是有专门的指令在后面默默支撑的。或者说，既然你把这些内存区域给硬生生地“生”出来了，那么你就必须要对人家负责，你必须设计指令对这些区域进行管理。

2. 函数调用指令

函数调用指令集可以归入到“程序转移指令集”中，也可以单独拿出来谈。由于 Java 中

的函数类型比较丰富，因此必然要支持更多的函数调用方式。詹爷设计了多个函数调用指令，例如，`invokevirtual`、`invokeinterface`、`invokespecial`、`invokestatic` 和 `return` 等。这比硬件所支持的函数调用指令集要丰富一些。以 x86 为例，x86 中主要使用 `call` 指令和 `ret` 指令来保存现场和恢复现场，这往往会伴随 CPU 物理寄存器入栈和出栈。

JVM 没有物理寄存器，所以用操作数栈和 PC 寄存器来替代。注意，这里又提到了操作数栈哦，重要的事说三遍，何况这么重要的概念！后面会反反复复提到这个概念哟。JVM 保存现场和恢复现场的解决方案是向 Java 堆栈中压入一个栈帧，函数返回的时候从 Java 堆栈中弹出一个栈帧。

JVM 调用函数的时候，不能像 CPU 硬件那样，直接跳转就能找到对应的代码段。这是因为 Java 函数的代码并没有被存放到代码段中，而是被放在了一个 code 缓存中，每一个 Java 函数的代码块在这个 code 缓存中都会有一个索引位置，最终 JVM 会跳转到这个索引位置处执行 Java 函数调用。同时，Java 的函数一定是被封装在类中的，因此 JVM 在执行函数调用时，还需要通过类寻址等等一系列运算，最终才能定位这个入口。

如果你没有汇编基础，并且对 JVM 也从来没有了解过，那么你在看这段话的时候，一定会晕。如果看不懂，没关系，后面会对照 JVM 源码进行详细讲解，所以这里看不懂很正常哦，千万不要灰心哟！而能够看得懂的童鞋们，自然会“会心一笑”，我知道你此刻心里挺美。

3. 运算指令集

JVM 和运算相关的指令集主要有算术运算、位运算、比较运算、逻辑运算等，JVM 还为各种基本类型的运算提供不同的操作码；x86 也有算术运算、逻辑运算、位运算、比较运算，但是所有的操作都是直接针对寄存器中的二进制数进行的，不区分数据类型。

JVM 规范中常见的运算指令包括 `iadd`（对两个 int 型数据求和）、`isub`（对两个 int 型数据做减法）、`fadd`（对两个 float 浮点数进行求和）、`ddiv`（两个 double 双精度型数据相除）等。

4. 控制转移指令

与硬件 CPU 一样，JVM 规范也提供了常见的控制转移指令，例如 `switch` 分支选择指令、`if...else` 条件判断、`do...while` 循环、`for` 循环、`foreach` 循环、`return` 返回、`break` 中断循环、`continue` 继续循环。不多讲，大家都懂的。

5. 对象创建与类型转换指令

作为一门面向对象的语言，JVM 规范自然要提供一套创建对象的指令。在 Java 语法层面使用关键字 `new` 可以实例化一个对象，而对应的字节码指令也是 `new`。

JVM 规范还提供了“窄化类型转换”指令，与“窄化类型转换”指令相对的是“宽化类型转换”指令，只不过后者是 JVM 内部天生支持的，不需要另外使用指令。

除了以上这些指令，JVM 规范还提供了很多其他物理 CPU 所没有的指令，例如，抛出异常的指令，用于线程同步的指令，等等。

由此可见，Java 字节码指令真的很神奇，足够简单，但又足够强大。字节码指令是中间语言的一种实现手段，虽然肯定还有其他技术路径。字节码指令能够完成 Java 语言的各种功能，能够压栈和出栈，能够读写局部变量表，能够调用方法，也能够创建对象实例。而最关键的一点是，它是跨平台的。这就是它很神奇的根本原因。

1.5 本章总结

如果你已经认真地看到了这里，并且你基本能理解所讲内容，那么恭喜你，你不知不觉中已经弄明白了 JVM 最核心的部分——run engine（执行引擎）。

如果你本身就拥有汇编基础、C 语言编程基础，本章所举的例子你都能看明白，或者你在自己的机器上亲手实践了一部分示例程序，那么更要恭喜你，你基本可以毫无障碍地看明白整个 JVM 核心的执行引擎模块了。JVM 核心的执行引擎虽然纷繁复杂，但是那只不过是同类功能的简单堆砌，将同类项合并后，它的基本核心技术其实都是本章所述内容。

本章简单介绍了 Java 语言产生的历史背景。Java 语言所要解决的是如何能够不关注底层技术细节就能实现兼容性，詹爷给出的答案是使用中间语言，通过中间语言来实现跨平台兼容的目标。由于中间语言并不是本地机器指令，机器 CPU 无法直接识别，因此中间语言并不能直接由物理 CPU 运行，那怎么办呢？很简单，使用虚拟机来解释中间语言，将中间语言翻译成对应的本地机器指令。

将中间语言翻译成本地机器码的方式有很多种，例如使用 C/C++ 语言为每一个 Java 字节码指令写一个对应的实现函数。但是这种方式太低效了。而解决低效的一种机制就是直接将 Java 字节码指令翻译成本地机器指令，运行期直接由 Java 虚拟机调用对应的机器指令来执行，这种调用的机制主要就是依靠 CPU 所提供的 call 和 jmp 指令。

中间语言长啥样？外表长得确实不够“漂亮”，很多人看了一眼就不愿意继续看第二眼，但是它很有内涵，能量足够大，能够跨平台。它就是 Java 字节码指令集。该指令集就是中间语言。这个指令集是对硬件 CPU 指令集的抽象与再加工，能够满足 Java 开发的一切需要。

第 2 章

Java 执行引擎工作原理：方法调用

本章摘要

- ◎ JVM 如何进行方法调用
- ◎ JVM 如何分配方法栈
- ◎ JVM 如何取指
- ◎ JVM 如何执行逻辑运算

JVM 作为一款虚拟机，也必然要涉及计算机核心的 3 大功能。

1. 方法调用

方法作为程序组成的基本单元，作为原子指令的初步封装，计算机必须能够支持方法的调用。同样，Java 语言的原子指令是字节码，Java 方法是对字节码的封装，因此 JVM 必须支持对 Java 方法的调用。

2. 取指

这里的“取指”，是指取出指令。

还是那句话，方法是对原子指令的封装，计算机进入方法后，最终需要逐条取出这些指令并逐条执行。Java 方法也不例外，因此 JVM 进入 Java 方法后，也要能够模拟硬件 CPU，能够从 Java 方法中逐条取出字节码指令。

3. 运算

计算机取出指令后，就要根据指令进行相应的逻辑运算，实现指令的功能。JVM 作为虚拟机，也需要具备对 Java 字节码的运算能力。

本章主要分析 JVM 如何从内部调用 Java 方法。

2.1 方法调用

到目前为止，人类发明出了若干种编程语言，有的编程语言没有类概念，有的编程语言面向过程，但不管是哪种编程语言，至少都会包含函数的概念。通过函数将一个大的程序拆分成体积小、功能明确的一个个简短的函数，从而将一个复杂的大型问题分解成若干简单的小问题，由繁到简。虽然函数并不总是大型软件模块化的手段，但一定是模块化得以实现的基础，否则随便开发个稍微难一点的功能，一写就是几千、几万行代码，估计没几个人能看懂，更没几个人有耐心看。

同理，Java 程序最基本的组成单位是类，而 Java 类也是由一个个的函数所组成，在这一点上，Java 也玩不出什么花样。

有的编程语言由真实的物理机器运行，有的程序运行于虚拟机上。既然所有的编程语言都由函数组成，那么运行由这些编程语言所开发出来的程序的机器就必须能够执行函数调用，不管是物理机器还是虚拟机器。JVM 作为一款虚拟机，要想具备执行一个完整的 Java 程序的能力，就必定得具备执行单个 Java 函数的能力。而要具备执行 Java 函数的能力，首先必须得能执行函数调用。

经过前面的讨论我们知道，詹爷当年为了能够让 Java 这门编程语言兼容各种平台，最终使用了一个大招——在运行时将 Java 字节码指令动态翻译成本地机器指令，从而既能获取兼容性，又能获取很高的运行效率。因此，JVM 实际上最后调用的并不是真正的 Java 函数，而是其对应的一堆机器指令。那么 JVM 究竟是怎么做到直接调用机器指令的呢？要研究清楚这个问题，必定要先弄清真实的物理机器是如何调用机器指令的。下面让我们简单了解一下真实的物理机器执行函数调用的机制。

2.1.1 真实的机器调用

本节主要通过一个汇编程序讲解一些真实的机器调用原理，若有道友看不懂也没有关系，多看几遍就懂了（哈哈）。想研究 JVM 的执行引擎原理，汇编这道坎必须得过，别无他法。

真实的机器指令调用机制涉及的知识比较多，例如，现场保存、堆栈分配、参数传递，等等。我们不需要知道那么专业的基础知识，只需要了解大体的原理，了解了这些原理，再理解 JVM 的函数调用就会变得简单了。废话少说，翠花，上菜！

下面这段程序是使用汇编编写的，程序功能很简单，对 2 个整数进行求和（注：由于直接写机器指令，相信没人能够看得懂，因此这里以机器指令的助记符——汇编语言进行演示）。例子如下：

清单：示例程序

作用：使用汇编进行求和

```
main:
    pushl    %ebp
    movl%esp, %ebp
    subl$32, %esp
    movl$5, 20(%esp)
    movl$3, 24(%esp)
    movl24(%esp), %eax
    movl%eax, 4(%esp)
    movl20(%esp), %eax
    movl%eax, (%esp)
    calladd
    movl%eax, 28(%esp)

    movl$0, %eax
    leave
    ret

add:
    subl$16, %esp
    movl12(%ebp), %eax
    movl8(%ebp), %edx
    addl%edx, %eax
    movl%eax, -4(%ebp)
    movl-4(%ebp), %eax
    leave
    ret
```

如果你不具备汇编基础知识，看不懂汇编程序，那么很难理解上面这段汇编程序的逻辑。不过看不懂没关系，只要能够明白其大意即可，不管哪种编程语言，其操作的无非是内存、数据，因此能够明白这段程序对内存做了什么，就可以了。

这段汇编程序想要实现的功能很简单，就是对两个整数进行求和。

分析一段程序，一般首先看该程序由哪些模块组成，分析汇编程序也是一样。这段汇编程

序在代码段中定义了两个标号，一个是 `main` 标号，一个是 `add` 标号。汇编语言中的标号类似于 C 语言中函数的概念，这里我们就当成函数好了。那么这段汇编程序中就定义了两个函数，一个是 `main()` 函数，一个是 `add()` 函数。看到这里，也许聪明的你很快就猜到，这段程序是不是在 `main()` 函数中定义了两个变量，然后传递给 `add()` 函数，由 `add()` 函数执行加法运算呢？恭喜你，猜对了。

既然明白了程序的大体意思，下面来一起分析下具体的算法。在这个过程中，如果你对汇编语法不怎么了解，也没关系，我们只需要关注最终内存是怎样变化的就可以了。

1. `main()` 函数详解

首先看 `main()` 函数。我们先整体解释下 `main()` 函数。

清单：示例程序

作用：使用汇编编写一段求和程序

```
main:
    // 保存调用者栈基地址，并为 main() 函数分配新栈空间
    pushl    %ebp
    movl    %esp, %ebp
    subl    $32, %esp           //这里就是分配新栈，一共 32 个字节

    // 初始化两个数据，一个是 5，一个是 3
    movl    $5, 20(%esp)
    movl    $3, 24(%esp)

    // 压栈：将 5 和 3 压入栈中
    movl    24(%esp), %eax
    movl    %eax, 4(%esp)
    movl    20(%esp), %eax
    movl    %eax, (%esp)

    // 调用 add() 函数
    call    add
    movl    %eax, 28(%esp) //得到 add() 函数的返回结果

    //返回
    movl    $0, %eax
    leave
    ret
```

看过 `main()` 函数的代码注释后，我们知道，`main()` 函数一共包含 5 步：保存调用者栈基地址，初始化数据，压栈，函数调用和返回。下面分别分析这 5 步过程。

1) 保存栈基并分配新栈

首先看第一步，main()函数从下面两条指令开始执行：

```
pushl    %ebp
movl %esp, %ebp
```

pushl %ebp 就是保存调用者的栈基地址。调用者是谁？谁能调用一个程序的主函数？当然是操作系统啦。movl %esp,%ebp 将调用者的栈基地址指向其栈顶。这两句是所有函数调用时都必定会执行的指令。add()函数的开头也是这两条指令。待会儿讲 add()函数调用时会再讲到它们，如果你不懂，可以先跳过。

执行完上面两条指令后，main()函数接下来执行下面这条指令：

```
subl $32, %esp
```

这条指令是干嘛用的？大家整天在讲分配栈空间，分配栈空间，这条指令就是干这事的。所以对于物理机器而言，分配堆栈空间非常容易，就一句话的事。这条指令中的 subl 表示减，指令中的 %esp 表示当前栈顶。整条指令的含义是：将当前栈顶减去 32 字节的长度。为什么是减，而不是加呢？这是因为在 Linux 平台上，栈是向下增长的，从内存的高地址往低地址方向增长，因此每次调用一个新的函数时，需要为新的函数分配栈空间，新函数的栈顶相对于调用者函数的栈顶，内存地址一定是低位方向，因此新函数的栈顶总是通过对调用者函数的栈顶做减法而计算出来。

执行了这条指令后，main()函数就有了自己的方法栈啦，栈空间大小是 32 字节，一个字节包含 8 个二进制位，如果一个 int 类型的整数包含 4 字节的话，那么 main()函数的方法栈就一共可以容下 8 个 int 类型的数据（如图 2.1 所示）。

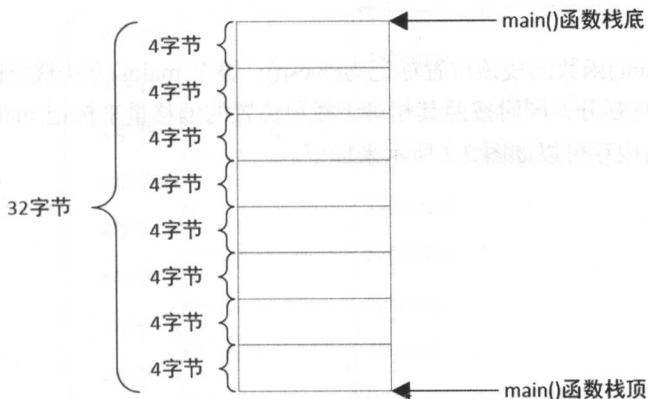


图 2.1 main()函数初始化时的堆栈

注：由于 `main()` 的方法栈空间刚刚分配，还没有往里面加任何东西，因此此时的栈是空的，什么都没有。同时，系统为 `main()` 函数一共分配了 32 个字节，在 64 位平台上，一个 `int` 型数据占 4 个字节，因此这里将 `main()` 方法栈以 4 字节为单位进行划分，一共划分成 8 块，每一块代表 4 个字节大小的空间。

`main()` 函数执行完上面这 3 条指令，便完成了调用者栈基地址的保存和自身栈空间的分配。

2) 初始化数据

`main()` 函数接下来执行下面 2 条指令：

```
movl$5, 20(%esp)
movl$3, 24(%esp)
```

这两条指令的含义是：分别将 5 和 3 这两个整数保存到 `main()` 栈中，其中 `20(%esp)` 表示当前栈顶（即 `esp` 寄存器当前所指向的内存地址）往上移动 20 字节位置，数据 5 就被保存在这里。由于 `main()` 函数的栈空间一共有 32 字节那么大，因此从 `main()` 方法栈的栈顶往上移动 20 字节后的位置，依然在 `main()` 的方法栈内。同理，整数 3 被保存到了 `main()` 函数栈顶往上偏移 24 字节处的位置。由于一个整数占用 4 字节（在 64 位平台上，本书中所有示例均在 64 位平台上测试，因此本文所有的 `int` 型数据默认都占 4 字节，下文不再赘述），因此 5 和 3 被分别保存到 `main()` 方法栈顶往上偏移 5 个整数和 6 个整数的位置。

语言描述让人理解起来总是很费电，还是图来得直观些。下面我们就通过图示来描述偏移位置。

在使用图示来描述内存位置之前，我们先研究一下真实的物理机器上是如何进行内存定位的。

如果我们将 `main()` 函数的栈顶位置标记为 `(%esp)`，整个 `main()` 方法栈空间的 32 字节，按照每 4 字节为单元进行划分，同时按照其相对于栈顶位置的偏移量来标记 `main()` 的方法栈，那么 `main()` 函数的方法栈内存可以如图 2.2 所示来标记。

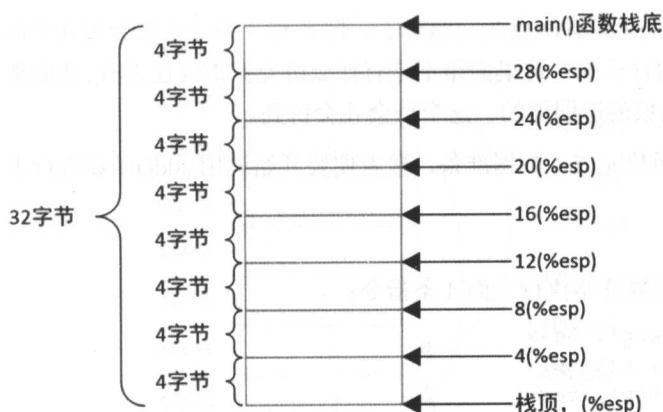


图 2.2 main()函数堆栈存储单元的相对位置

这种将方法栈内存位置按照栈顶偏移量进行标记的方法，是整个计算机的理论基础，直接影响了编程语言的模块划分方法。如果没有这种方法，编译器很难在编译期就确定各个变量的位置，更不用谈各种基于编译原理的高级应用了。

其实这种标记方法在现实世界中也是很常见的。对于地球上任意一个点，人类可以使用两种方式标记其位置，一种是绝对定位，一种是相对定位。例如以某个学校为例，我们可以说这个学校在经度 30° 和纬度 60° ，也可以说这个学校在前方红绿灯右转 300 米。物理机器对方法内部的局部变量的定位就类似于对学校的“红绿灯右转 300 米”的相对定位法。

理解了相对定位方法后，我们再来看看 5 和 3 这两个数据在 main() 方法栈中的位置，如图 2.3 所示。

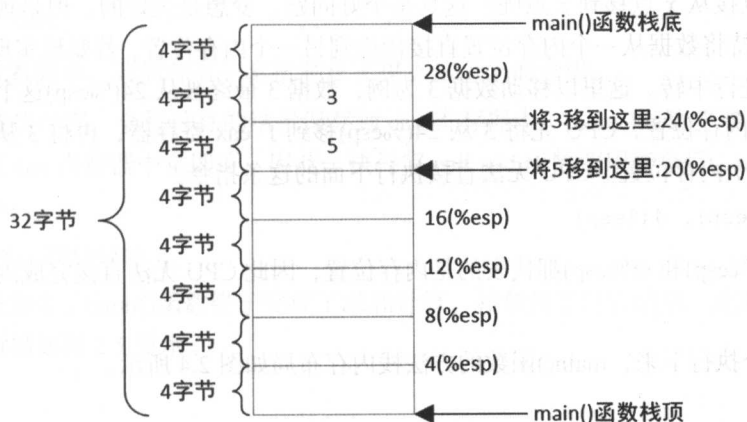


图 2.3 完成局部变量初始化的 main() 方法堆栈布局

也许有些完美主义者看出点儿问题来了，即 5 和 3 为什么被分配在中间的位置，上边和下边哪都不挨，这是为什么呢？这里面是不是有什么讲究？讲究还真有，栈底那个位置即 `28(%esp)` 是留给调用 `add()` 函数的返回值的，这个待会儿会讲到。

现在，`main()` 函数完成了数据准备，接着便要开始调用 `add()` 函数进行求和了。

3) 压栈

接着，`main()` 函数开始执行下面 4 条指令：

```
movl 24(%esp), %eax
movl %eax, 4(%esp)
movl 20(%esp), %eax
movl %eax, (%esp)
```

这 4 条指令主要作用是“压栈”。前两条指令是把数据 3 压栈，后两条指令是把数据 5 压栈。

先看 `movl 24(%esp), %eax` 这条指令，该指令将 `24(%esp)` 处的内存值传送到 `eax` 寄存器中。`24(%esp)` 处的内存值是什么呢？就是刚才第 2 步中保存的整数 3。接着 `movl %eax, 4(%esp)` 这条指令又将 `eax` 寄存器中的值传送到了 `4(%esp)` 这个地方。如果将这两条指令连着看，你会发现这里进行的数据传送的路径是： $x \rightarrow y$, $y \rightarrow z$ ，使用小学数学进行推理，可以推出： $x \rightarrow z$ ，即 x 处的内存值被传送到了 z 处。因此，最终的效果是：整数 3 被从 `24(%esp)` 这个相对于栈顶的偏移位置，转移到了 `4(%esp)` 这个偏移位置。

同样的道理，后面两条指令的最终效果是：整数 5 被从 `20(%esp)` 这个相对于栈顶的偏移位置，转移到了 `(%esp)` 这个偏移位置。`(%esp)` 是哪个位置？请相信你的第一直觉，就是栈顶。

也许好奇心重的同学会想，既然 CPU 可以将一个数据从 x 点移到 y 点，再从 y 点移到 z 点，那么为什么不直接从 x 点移到 z 点呢？这真是个好问题。梦想是美好的，但是现实是残酷的，因为 CPU 不支持将数据从一个内存位置直接传送到另一个内存位置，若要想实现这个效果，必须使用寄存器进行中转。这里以移动数据 3 为例，数据 3 最终被从 `24(%esp)` 这个内存位置移到了 `4(%esp)` 这个内存位置，CPU 先将 3 从 `24(%esp)` 移到了 `eax` 寄存器，再将 3 从 `eax` 寄存器移到了 `4(%esp)` 这个内存位置。CPU 无法直接执行下面的这条指令：

```
movl 24(%esp), 4(%esp)
```

只因为 `24(%esp)` 和 `4(%esp)` 都代表的是内存位置，因此 CPU 无法直接完成内存之间的数据传送。

这 4 条指令执行下来，`main()` 函数的方法栈内存布局如图 2.4 所示。

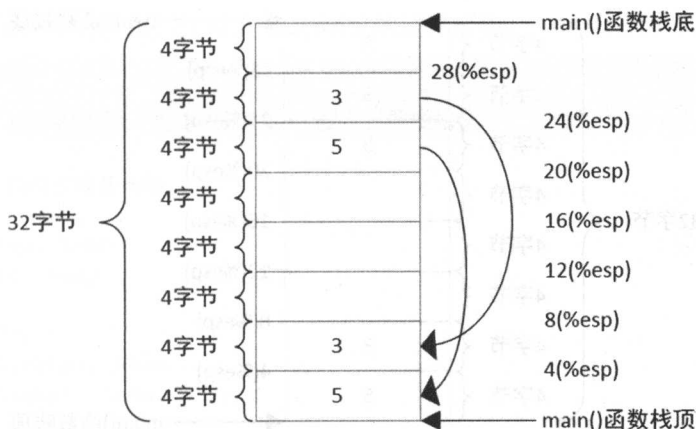


图 2.4 完成 add()参数复制后的 main()函数堆栈布局

一般而言，往栈顶传送数据的行为叫做“压栈”，这里先后将 3 和 5 都放到了栈顶处，因此这都是压栈。

`main()`函数为何要压栈呢？那是因为 `main()`函数即将要进行函数调用了。

真实的物理机器，在发起函数调用之前，必定要进行压栈。压栈的目的是为了传参。

`main()`函数在这里压栈的两个数据，将会被 `add()`函数读取到。

4) 函数调用

压完了栈，`main()`函数终于开始进行函数调用了。对于物理机器而言，函数调用特别简单，就一条指令：

```
call    add
```

但是，看着简单，背后却有一套机制在支撑。这个下面会讲。

`add()`函数执行完，会将计算的结果保存至 `eax` 寄存器中。`main()`函数要取得 `add()`函数返回值，便直接从 `eax` 寄存器中拿即可。因此，执行完 `call add` 函数调用指令后，`main()`函数接着调用下面的指令：

```
movl%eax, 28(%esp)
```

通过这条指令，`main()`函数终于完成了求和计算，并拿到了计算结果。此时，`main()`函数的方法栈内存布局如图 2.5 所示。

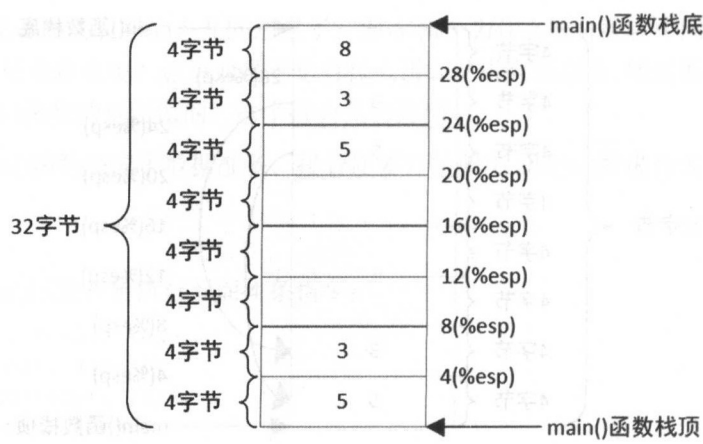


图 2.5 完成 add()调用后的 main()函数堆栈布局

到了这里，你会发现其实这样的内存布局还是挺美的，什么叫美？历史上无论是科学大牛还是艺术巨匠都告诉我们：对称的，才是美的。你看现在的 main()方法栈的内部布局图，就挺对称，上面的数据都挨着栈底，下面的数据都挨着栈顶。之所以会这样，全是编译器的功劳。编译器会将一个方法内的局部变量分配在靠近栈底的位置，而将传递的参数分配在靠近栈顶的位置。

5) 返回

函数返回很简单，将返回值保存到 `eax` 寄存器中，然后执行两条例行返回指令便大功告成。main()函数的返回指令就是下面这 3 条：

```
movl $0, %eax
leave
ret
```

好了，到此为止，我们完整地分析了 main()函数的执行过程以及栈内存的布局演变。在分析的过程中，顺带着了解了 main()函数为了调用 add()函数而做的准备。在本示例中，main()函数为了调用 add()函数，主要是将入参进行了“压栈”操作，这样在 add()函数内部才能取到参数并进行求和运算。

但是，除了压栈外，其实系统还要做一部分工作，才能完成最终的方法调用。下面我们通过分析 add()函数的执行过程，来了解系统工作的机制。

2. add()函数详解

如果你对上面 main()函数的执行原理已经比较熟悉，那么理解 add()函数的运行机制就容易

了。在分析 add() 函数之前，我们先看下 add() 函数的整体逻辑：

清单：示例程序

作用：使用 C 程序提供的语法糖修改 CS:IP 段寄存器的指向

```
add:
    // 保存调用者栈基地址
    pushl    %ebp
    movl    %esp, %ebp
    subl    $16, %esp

    // 获取入参
    movl    12(%ebp), %eax
    movl    8(%ebp), %edx

    // 执行运算
    addl    %edx, %eax
    movl    %eax, -4(%ebp)

    // 返回
    movl    -4(%ebp), %eax
    leave
    ret
```

可以看到，add() 函数总体上分为 4 步：保存调用者栈基地址，读取入参，执行运算，返回。下面我们逐一分析过程。

1) 保存调用者栈基地址

add() 函数也是以下面两条指令开始：

```
pushl    %ebp
movl     %esp, %ebp
```

这一步大家应该很熟悉，刚才在分析 main() 函数时就已经讲过。这两条指令主要是保存调用者栈基地址。这里再啰嗦一次，物理机器在执行函数调用时，被调用者总是要保存调用者栈基地址。这是因为 esp 和 ebp 这两个寄存器接下来要指向被调用者的栈基地址和栈顶，这两个寄存器原本保存的是调用者的栈基地址和栈顶地址，现在即将被修改，如果不保存起来，那么当被调用者函数执行完成，程序流返回到调用者流程中时，物理机器将无法恢复调用者的栈基和栈顶，从而导致程序无法继续执行下去。

add() 函数第 3 条指令是：

```
subl     $16, %esp
```

这条指令大家应该很熟悉，就是分配栈空间。为谁分配？当然是为当前函数 add()。分配多大空间？这条指令中的 16 来指定。不过要注意，这里指 16 字节，而不是 16 个二进制位。

现在，让我们看看方法栈空间的内存布局。由于 `add()` 函数的方法栈是在调用方 `main()` 函数的方法栈空间基础上往下增长的，并且 `add()` 方法栈与 `main()` 方法栈连在一起，因此现在我们同时看 `main()` 和 `add()` 两个函数的方法栈。如图 2.6 所示。

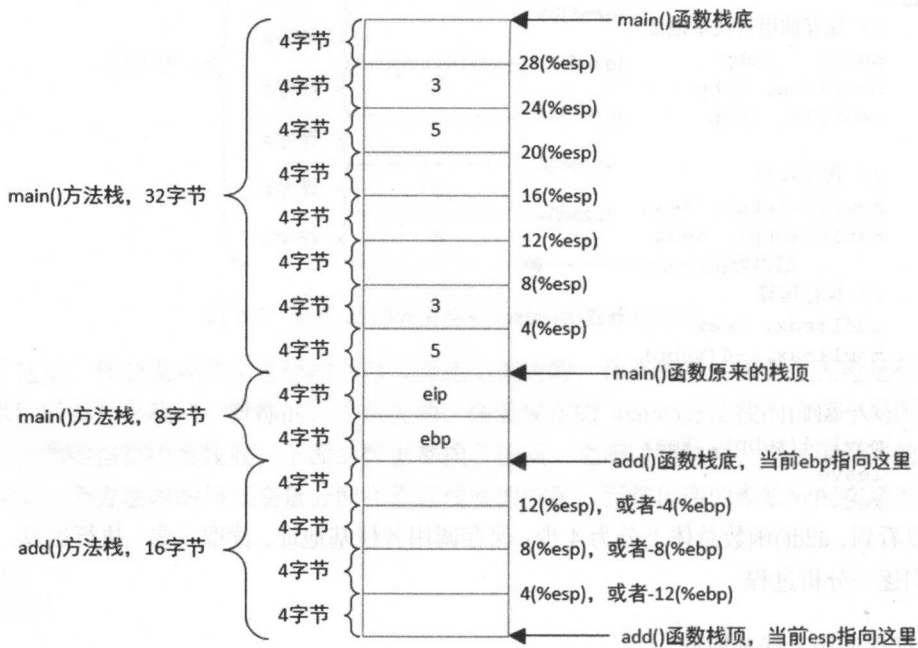


图 2.6 `main()` 函数调用 `add()` 函数时的整体堆栈布局

现在这张栈内存布局图同时包含 `main()` 函数和 `add()` 函数的方法栈。`main()` 方法栈大小为 32 字节，`add()` 方法栈大小为 16 字节。但是奇怪的是，在 `main()` 和 `add()` 方法栈的中间，竟然多了 8 个字节的空间。这 8 个字节的空间是从哪里冒出来的呢？答案很简单，在 `main()` 函数执行 `call add` 指令时，物理机器自动往栈顶压了一个数值——`eip`。前面我们扫盲汇编基础知识时讲过，CPU 所执行的指令位置由 `CS:IP` 这 2 个段寄存器共同决定，这里的 `eip` 就是 IP 寄存器。物理机器为何要将这个寄存器的值入栈呢？这主要是为了让 `main()` 函数执行完 `call` 调用回来之后，能够继续处理 `main()` 函数中接下来的指令。我们看 `main()` 函数中 `call add` 指令的上下文，如果没有 `call add` 指令，即如果 `main()` 函数不调用 `add()` 函数，那么 `main()` 函数执行完 `call add` 这条指令的上一条指令 `movl %eax, (%esp)` 之后，`main()` 函数应该接着执行 `call add` 这条指令的后面一条指令 `movl %eax, 28(%esp)`，同时，`main()` 函数在执行 `movl %eax, 28(%esp)` 这条指令之前，`eip` 寄存器需要先指向这条指令，这样 CPU 才能读取到这条指令并执行。但是现在，`main()` 函数在执行这条指令之前，先调用了 `add()` 函数，那么 `eip` 就会指向 `add()` 函数里面的指令内存位置（这一步是

物理机器自动执行的),那么当 add()函数执行完其最后一条指令后,物理机器怎么知道接下来要把 eip 寄存器指向哪里,或者说物理机器怎么知道接下来要执行哪里的指令呢?所谓的函数只是人类进行的模块划分,物理机器可不知道有这些东东,物理机器更不会自动记忆当前函数被哪个函数调用,更不会在执行完当前函数后,自动跳转到当前函数的调用者函数中去执行。所以,执行完一个函数,我们不去修改 eip 寄存器的值,那么物理机器根本就不知道接下来应该怎么做。基于这样的原因,在执行函数调用时,CPU 设计者在里面加了一个功能,即在物理机器执行 call 指令时,自动将当前 eip 寄存器入栈。而当被调用者执行完之后,物理机器再自动将 eip 出栈,这样,执行完被调用函数之后,物理机器会接着执行调用者的后续指令。

刚刚解释了 main()和 add()函数中间多出来的 8 字节中的 eip, eip 占 4 字节,因此还有 4 字节的空间。我们看图 2.6 可以知道,这剩下的 4 字节空间存放着 ebp 的值。这个值是在执行 add()函数时入栈的。我们看,add()函数的开头第一条指令就是 pushl %ebp,这里显式执行了 push 入栈操作。

经过上面对 add()函数的初步分析,我们可以得出以下结论:

- ◎ 物理机器执行 call 函数调用时,机器会自动将 eip 入栈。
- ◎ 物理机器执行函数调用时,被调用方需要手动将 ebp 入栈。

add()函数执行完开头的 3 条指令后,机器开始进入 add()函数域,接下来开始执行 add()函数里面真正的逻辑运算。

2) 读取入参

add()函数的第 4 和第 5 这两条指令为读取入参指令,先看指令:

```
movl 12(%ebp), %eax
movl 8(%ebp), %edx
```

第一条指令是 movl 12(%ebp), %eax,这条指令中使用了 2 个寄存器:ebp 和 eax,其中 ebp 寄存器的用途与 esp 一样专一,只用于标识栈底位置。对于 12(%ebp)这种写法我们已经比较熟了,表示从 ebp 寄存器所指向的内存地址往高地址方向偏移 12 字节。由于在 add()函数的一开始执行了 movl %esp, %ebp 指令,因此此时 ebp 寄存器已经指向了原来 main()函数的栈顶。第一条指令合起来的意思就是,从 add()函数栈底向上偏移 12 字节的位置取出数据(占 4 字节),将该数据传送给 eax 寄存器。

同理,第二条指令的意思是,从 add()函数栈底向上偏移 8 字节的位置取出数据(占 4 字节),将该数据传送给 edx 寄存器。

通过这两条指令,add()函数成功从 main()函数中获取到了两个入参。

为什么要从这两个位置读取入参呢？这是因为 `main()` 函数在把两个入参压栈后，执行 `call` 函数调用指令，由于系统会继续将 `eip` 和 `esp` 压栈，这两个数据共占 8 字节，因此导致 `add()` 栈顶与 `main()` 函数中压栈的两个参数之间隔着 8 字节的距离，因此 `add()` 函数要分别从这两个位置获取入参。

我们在堆栈内存图上将这两个入参的位置标记出来，如图 2.7 所示。

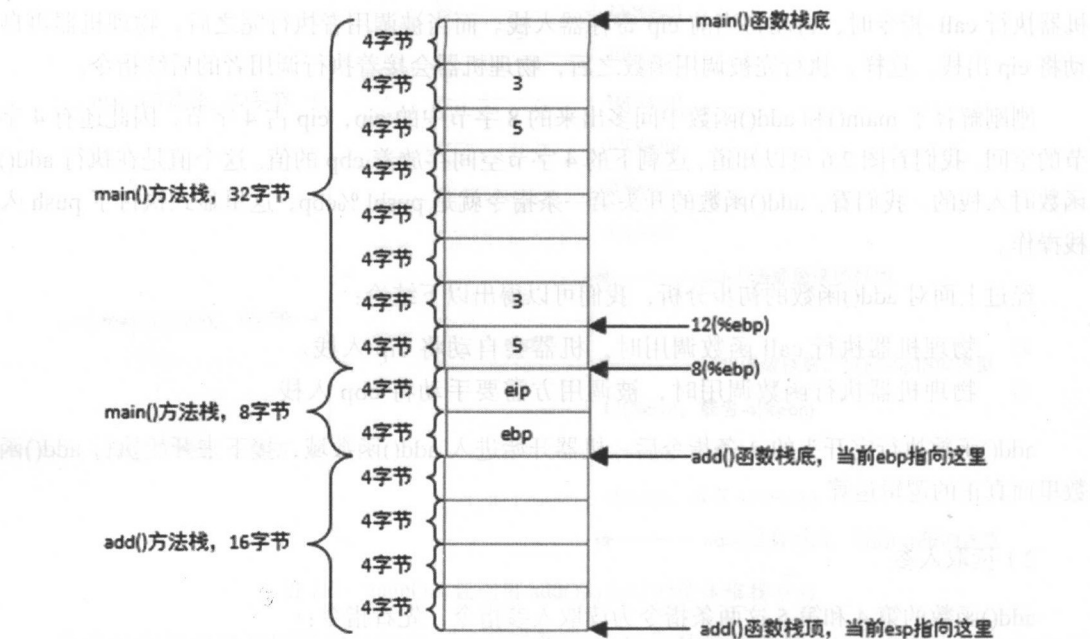


图 2.7 `main()` 函数调用 `add()` 函数时所压栈的入参位置

之前在 `main()` 方法中执行参数压栈指令时，当时压栈的两个位置分别是 `(%esp)` 和 `4(%esp)`，而现在这两个位置的标记变成 `8(%ebp)` 和 `12(%ebp)`，由此可见，随着堆栈寄存器 `ebp` 和 `esp` 所指向位置的变化，方法栈中同一个内存位置的偏移量也随之改变。同时，对于压栈的入参，既可以从通过相对于调用者函数的栈顶的偏移量来相对定位，也可以通过相对于被调用者函数的栈底的偏移量来相对定位。当然，如果你愿意，你也可以通过相对于调用者函数的栈顶偏移位置来相对定位。总之，对方法栈内存的定位手段是灵活的，可以选择不同的参考系，不同的定位基准决定了不同的偏移量和定位方法。但是对于被调用者函数的方法栈内的数据，却不能以调用者函数为基准通过偏移量获取，因为此时被调用函数尚未分配方法栈空间，根本取不到数据，甚至会取到错误的数据。下面我们在 `add()` 指令中将会遇到不同的相对定位方式。

3) 执行运算

add()函数主要功能是求和，而求和运算是物理机器的最基本的功能之一，因此物理机器提供了一条指令专门用于求和：add。add()函数中求和的指令如下：

```
addl%edx, %eax
```

这条指令的含义是，将 edx 寄存器中的值与 eax 寄存器中的值相加，相加结果保存到 eax 寄存器中。

在 add()函数执行本指令之前，已经通过读取入参指令将 main()函数所传递过来的两个参数分别读取到了 eax 和 edx 这两个寄存器中，因此对这两个寄存器执行求和操作，就相当于对 main()函数传递过来的 2 个入参执行求和。

执行完求和运算，add()函数接着执行 movl %eax, -4(%ebp)这条指令。这条指令的作用是把 eax 寄存器中的值转移到栈地址往下偏移 4 个字节的位置。这个位置是哪里呢？其实就是 add()函数的方法栈内的第一个位置。add()函数执行到现在，虽然分配了 16 字节的空间，但是一直还未使用过。现在终于用到了。此时 eax 寄存器中存放的是什么数据呢？就是刚刚执行 add 求和指令的结果。由此可知，add()将求和的结果保存在了其方法栈的第一个位置。此时整体堆栈内存布局如图 2.8 所示。

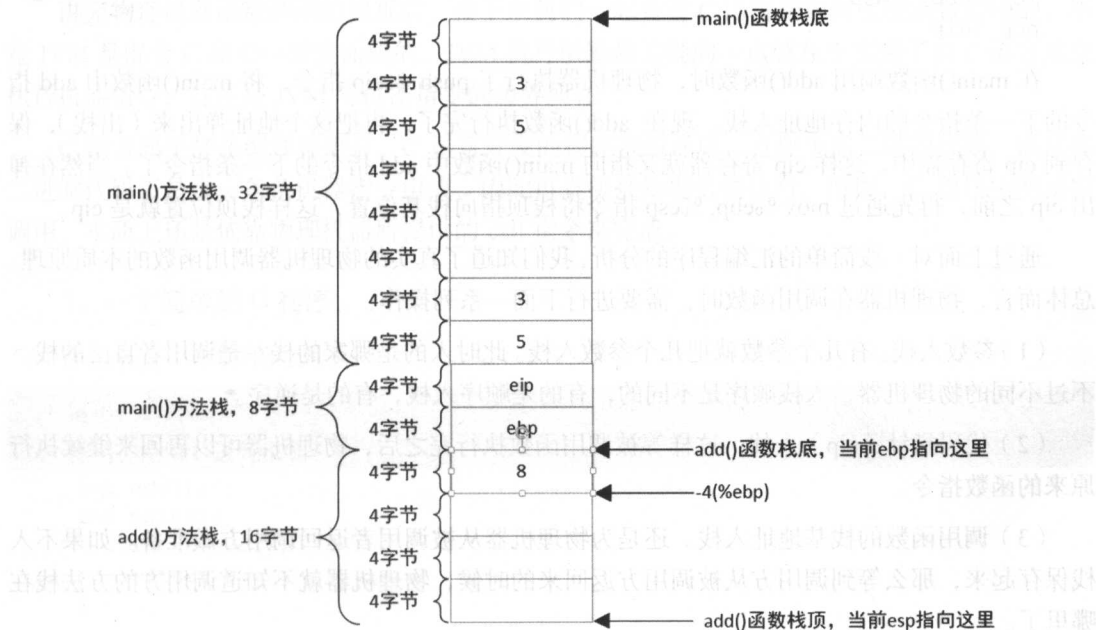


图 2.8 add()方法完成求和时的堆栈布局

这里,我们可以看出,对一个具体的方法栈中的某个位置而言,我们既可以通过 `ebp` (栈底)进行相对定位,也可以通过 `esp` (栈顶)进行相对定位。例如这里,对于 `add()` 函数方法栈的第一个位置,既可以通过 `-4(%ebp)` 相对定位,也可以通过 `12(%esp)` 相对定位。`esp` 或 `ebp` 前面的正负符号表示相对基准位置是向上偏移还是向下偏移。

4) 返回

执行完求和运算后, `add()` 函数的使命便完成了,接下来该返回了。函数返回的一般逻辑是,如果有返回值,就把返回值放在 `eax` 寄存器中,然后执行 `leave` 和 `ret` 指令。如果没有返回值,则直接执行 `leave` 和 `ret` 指令。当然这里多啰嗦一点, `leave` 和 `ret` 指令其实封装了好几个命令,以后大家看到其他汇编函数返回时执行的不是这两条命令,也不要感到太意外。指令是死的,人是活的。有的人喜欢使用封装好的指令,而有的人则喜欢使用最原始的指令,就好比巫师念起那最古老的咒语一般。

至此, `add()` 函数全部执行结束,程序流程又返回到了 `main()` 函数中。程序流为何能够返回到 `main()` 函数中呢?前面讲过,物理机器要想执行某个函数中的指令,必须先把 `ip` 段寄存器指向那个函数。`ip` 指向哪里,物理机器就执行到哪里。在 `add()` 函数返回的 `leave` 指令中,其实顺带着执行了下面这两条指令:

```
mov %ebp, %esp
pop %eip
```

在 `main()` 函数调用 `add()` 函数时,物理机器执行了 `push %eip` 指令,将 `main()` 函数中 `add` 指令的下一条指令的内存地址入栈。现在 `add()` 函数执行完了,再把这个地址弹出来(出栈),保存到 `eip` 寄存器中,这样 `eip` 寄存器就又指向 `main()` 函数中 `add` 指令的下一条指令了。当然在弹出 `eip` 之前,得先通过 `mov %ebp, %esp` 指令将栈顶指向栈基位置,这样栈顶位置就是 `eip`。

通过上面对一段简单的汇编程序的分析,我们知道了真实的物理机器调用函数的本质原理。总体而言,物理机器在调用函数时,需要进行下面一系列操作:

(1) 参数入栈。有几个参数就把几个参数入栈。此时入的是哪家的栈?是调用者自己的栈。不过不同的物理机器,入栈顺序是不同的,有的是顺序入栈,有的是逆序。

(2) 代码指针(`eip`)入栈,这样等被调用函数执行完之后,物理机器可以再回来继续执行原来的函数指令。

(3) 调用函数的栈基地址入栈。还是为物理机器从被调用者返回调用方做准备。如果不入栈保存起来,那么等到调用方从被调用方返回来的时候,物理机器就不知道调用方的方法栈在哪里了。

(4) 为被调用方分配方法栈空间。每一个函数都有自己的栈空间,如同地球上的每一个人

都有自己的小窝。

通过对这段汇编程序的分析可知，物理机器在执行程序时，将程序划分成若干函数，每个函数都对应有一段机器码。一段程序的机器码都放在一块连续的内存中，这块内存叫做代码段。物理机器为每一个函数分配一个方法栈，方法栈与代码段在地址上没有任何关系，并且只有当物理机器执行到某个函数时，才会为其分配方法栈，否则就不会分配。函数通过自身的机器指令遥控其对应的方法栈，可以往里面放入数值，也可以将数值移动到其他地方，也可以从里面读取数据，也可以从调用者的方法栈里取值。通过一条条指令和一个个栈，物理机器得以运行完整个程序。

知道了物理机器调用函数的这些秘密，我们再分析 JVM 的函数调用机制就不会感到高深莫测了。其实 JVM 也是将 Java 函数所对应的机器指令专门存储在内存的一块区域上，同时为每一个 Java 函数分配了方法栈。但是在真正了解 JVM 的函数调用机制之前，我们还有最关键的一步需要了解。而要了解这一步，我们需要先看看同样是高级语言的 C 语言是如何执行函数调用的。

2.1.2 C 语言函数调用

讲完物理机器函数调用的原理后，接下来我们一起来看看 C 语言是如何实现函数调用的。毕竟 JVM 是混合 C 和 C++ 开发而成的，JVM 执行引擎最关键的一点就在于实现了由 C 语言动态执行机器指令，这正是 JVM 与机器指令的边界所在。

C 语言属于静态编译型语言，C 语言开发的程序被编译后，直接生成二进制代码，而这些二进制代码正是由一条条机器指令组成，因此可直接被物理机器执行。所以，C 程序中的函数调用，本质上还是依靠物理机器所提供的 call 指令来完成。

1. 一个简单的 C 程序

首先来看一个简单的例子：

清单：示例程序

作用：使用 C 进行求和

```
int add();
int main(){
    int c=add();
    return 0;
}
```

```
int add(){
```

```
int z=1+2;
return z;
```

本例十分简单，add()函数主要完成两个整数的累加并返回累加结果，main()函数调用 add()函数并返回 0。

本例中的 add()函数没有参数，因此不涉及调用者与被调用者之间的参数传递。

将这段 C 程序编译成汇编程序，如下所示：

清单：示例程序

作用：C 程序对应的汇编

```
main:
    pushl    %ebp
    movl%esp, %ebp
    andl$-16, %esp
    subl$16, %esp
```

```
    calladd
    movl$0, %eax
    leave
    ret
```

```
add:
    pushl    %ebp
    movl%esp, %ebp
    subl$16, %esp
    movl$3, -4(%ebp)
    movl-4(%ebp), %eax
    leave
    ret
```

仔细观察这段汇编程序，你会发现这段汇编程序有两个标号，分别是 main 和 add。而巧合的是，源程序 C 程序中恰好包含了 main()函数和 add()函数。其实，这不是巧合，而是编译器处理的结果。编译器在编译 C 程序时，会将 C 程序中的函数名处理成汇编程序中的标号。

事实上，这段汇编程序中的 main 和 add 正是汇编程序的 2 个代码段。汇编程序由一个一个代码段组成，如同 C 程序由一个一个函数组成一样。

有了标号，汇编程序就能执行函数调用，即执行 call 指令。可以看到，在这段汇编程序中，main 代码段所对应的汇编代码中，有一条 calladd 指令，这正是汇编语言中的函数调用指令。因此，C 程序中的 main()函数调用 add()函数，在汇编程序中就转换成了 calladd。这就是 C 程序函数调用的秘密所在。

继续仔细分析这段汇编程序，我们会发现无论是 main 代码段，还是 add 代码段，一开始都包含下面 3 条指令：

```
pushl    %ebp
movl %esp, %ebp
andl $-16, %esp
```

不过 add 代码段一开始的第 3 条指令与 main 代码段的第 3 条指令有所不同，add 代码段的第 3 条指令是 `subl $16, %esp`，该指令其实与 main 代码段中的第 3 条指令 `andl $-16, %esp` 所表达的含义是相同的，那就是分配堆栈空间。

这 3 条指令的含义在上面讲解汇编函数调用机制时详细说明过，其作用是，保存调用者栈基址，并为新方法分配方法栈，这几乎成为汇编程序进行方法调用的标准定式。不过并非所有的汇编程序中的方法都会按照这样的顺序来执行，并且这 3 条指令也并不一定紧靠在一起，在本例中之所以会如此一致，那都是编译器处理后的结果。但是编译器是死的，而人是活的，在 JVM 中，很多汇编程序直接由人工编写，并非依靠编译器，因此会有所不同。不过总体而言，汇编程序在进入方法时，这 3 条指令是不会少的。

研究函数调用，堆栈分析是一定不可少的，这是物理机器实现方法调用的核心算法。在本例中，当 main() 函数执行到 `call add` 这条指令之前，物理机器会为 main() 函数分配方法栈，堆栈空间是 16 字节。main() 函数的方法栈内存布局如图 2-9 所示。

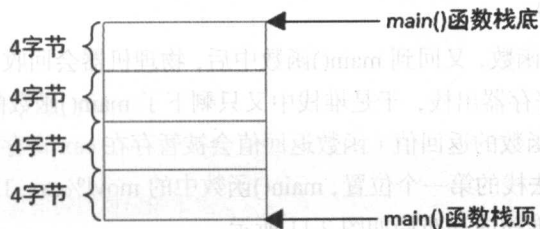


图 2.9 main() 函数初始堆栈布局

此时 main() 方法栈是空的，啥都没有。

当物理机器执行完 add() 函数的最后一条指令 `movl -4(%ebp), %eax` 时，堆栈内存布局如图 2.10 所示。

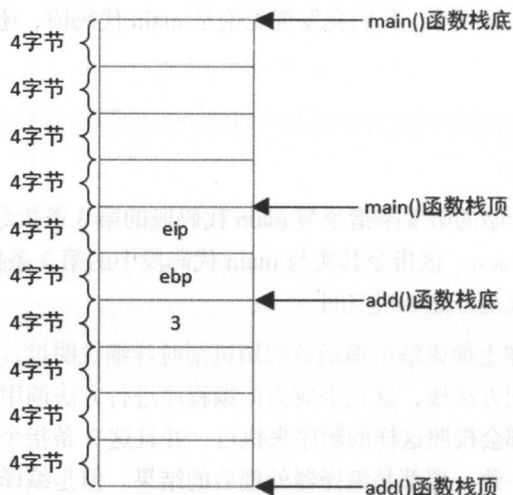


图 2.10 执行完 add()函数时的整体堆栈布局

可以发现,物理机器将 `ebp` 和 `ebp` 两个寄存器压入 `main()` 函数的栈顶(上一节讲过),同时为 `add()` 函数分配了 16 字节的堆栈空间。由于 `add()` 函数源程序的 `int z=1+2` 被编译器自动算出了结果 3,因此编译器直接将立即数 3 分配到了 `add()` 函数的方法栈中第一个位置,也即最靠近 `add()` 函数方法栈栈底的位置,分配的指令就是汇编程序中 `add` 代码段中的:

```
movl $3, -4(%ebp)
```

当程序执行完 `add()` 函数，又回到 `main()` 函数中后，物理机器会回收 `add()` 函数的堆栈空间，同时 `eip` 和 `ebp` 这两个寄存器出栈，于是堆栈中又只剩下了 `main()` 函数的内存。`main()` 函数会从 `eax` 寄存器中读取 `add()` 函数的返回值（函数返回值会被暂存在 `eax` 寄存器中，这是约定），并将其传送到 `main()` 函数方法栈的第一个位置，`main()` 函数中的 `movl%eax, 12(%esp)` 指令执行的正是这种数据传送。此时，堆栈内存布局如图 2.11 所示。

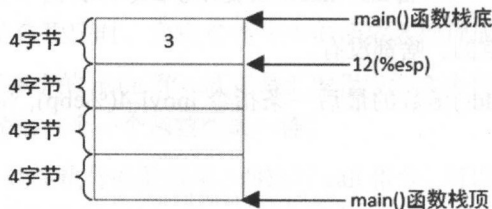


图 2.11 main()函数完成 add()调用之后的堆栈布局

可能有人会问，main()函数其实只需要分配 4 字节的堆栈空间，用于保存计算结果 3，可为何编译器为其分配了 16 字节的空间呢？这是因为这也是一种约定，就是内存对齐。在 32 位和 64 位机器上，堆栈内存都是按照 16 字节进行对齐的，多了不退，少了一定会补齐。之所以会有这种约定，道理很简单，就是为了能够对内存进行快速定位、快速整理回收。

2. 带参数的 C 程序

刚才所举例子中的 C 程序，并不涉及参数传递。下面我们将 C 程序稍微修改一下，使函数带有参数。

修改后的 C 程序如下：

清单：示例程序

作用：带有人参的 C 求和程序示例

```
int add(int a, int b);
int main(){
    int a = 5;
    int b = 3;
    int c=add(a, b);
    return 0;
}

int add(int a, int b){
    int z=1+2;
    return z;
}
```

将其编译成汇编程序，如下所示：

清单：示例程序

作用：带有人参的 C 求和程序对应的汇编

```
main:
    pushl    %ebp
    movl    %esp, %ebp
    andl    $-16, %esp
    subl    $32, %esp

    movl    $5, 20(%esp)
    movl    $3, 24(%esp)
    movl    24(%esp), %eax
    movl    %eax, 4(%esp)
    movl    20(%esp), %eax
    movl    %eax, (%esp)

    call    add

    ret
```

```

movl %eax, 28(%esp)
movl $0, %eax
leave
ret

```

```

add:
pushl   %ebp
movl %esp, %ebp
subl $16, %esp
movl $3, -4(%ebp)
movl -4(%ebp), %eax
leave
ret

```

上面这段汇编程序是不是看起来很熟悉？汇编程序中包含两个标段 `main` 和 `add`，`main` 可以向 `add` 传入两个整型参数，`add` 负责对两个人参进行求和并将累加结果返回给 `main`。其实这就是前文在分析物理机器函数调用时所举的例子。

前面那个例子其实就是这段 C 程序编译后的汇编程序。前文已经详细分析过这段汇编程序的逻辑以及执行过程中的堆栈内存布局及其变化，此处不再赘述。本节我们主要研究的是 C 程序的函数调用机制。

前面我们分析了没有参数传递场景下的 C 程序函数调用机制，这里我们再次温习一下有参数传递场景下的 C 程序函数的调用机制，主要包括以下 3 点。

1) 压栈

如同前文分析，本例中的 `main()` 函数在调用 `add()` 函数之前，会将两个人参压栈，压入栈之后，`add()` 就可以取得这两个人参。压入了谁的栈？当然是调用者的堆栈，即 `main()` 函数。

2) 参数传递顺序

对于 Linux 平台而言，调用者函数向被调用者函数传递参数时，采用逆向顺序压栈，即最后一个参数第一个压栈，而第一个参数最后压栈。

这种压栈顺序决定了被调用函数对入参进行寻址的方式和顺序。

3) 读取入参

`main()` 函数将入参压栈后，作为被调用者的 `add()` 函数怎么获取入参呢？前文分析过，是通过 `add()` 函数的栈基地址 `ebp` 的相对地址，从 `main()` 函数中读取入参。最后一个人参位置在 `8(%ebp)`，倒数第二个人参位置在 `12(%ebp)`，以此类推。

之所以要从相对 `ebp` 寄存器往上第 8 个存储位置开始寻址，是因为调用者与被调用者函数堆栈之间隔着 `eip` 和 `ebp` 这两个寄存器值。

以上3点应当牢记，这对后面分析JVM的执行引擎机制大有裨益。

总体而言，本节主要通过汇编程序和C语言程序演示了真实的物理机器执行函数调用时的原理，如果不把这些内容弄清楚，则在研究JVM执行引擎时会寸步难行。在真实的物理机器上，执行函数调用时主要包含以下几个步骤：

- (1) 保存调用者栈基地址（即当前栈基地址入栈），当前IP寄存器入栈（即调用者中的下一条指令地址入栈）。
- (2) 调用函数时，在x86平台上，参数从右到左依次入栈。
- (3) 一个方法所分配的栈空间大小，取决于该方法内部的局部变量空间、为被调用者所传递的入参大小。
- (4) 被调用者在接收入参时，从8(%ebp)处开始，往上逐个获取每一个入参参数。
- (5) 被调用者将返回结果保存在eax寄存器中，调用者从该寄存器中获取返回值。

2.2 JVM 的函数调用机制

前文我们一起学习了物理机器的函数调用机制和C语言的函数调用机制，本质上这两者的函数调用机制是相同的，因为C语言是静态编译型语言，编译后就变成了能够直接被物理机器执行的二进制代码，所以C程序中的函数调用最终还是直接依赖物理机器的函数调用指令。

现在如果让我们来开发类似JVM这样的一款解释型虚拟机，首先要解决的问题就是函数调用，你会怎么做呢？

JVM是用C和C++语言编写的一款软件，当JVM执行Java函数时，实际上是执行了一段汇编代码，换言之，这中间一定存在一个边界，在边界处，边界的一边是C程序，边界的另一边直接是机器指令，C语言要能够直接执行机器指令。例如，如果你编写了下面一段非常简单的Java代码：

清单：示例Java程序

作用：演示Java语言求和

```
public class Test {
    public static void main(String[] args) {
        add(5,8);
    }

    public static int add(int a, int b) {
        int c = a + b;
    }
}
```

```

        int d=c+9;
        return d;
    }
}

```

这段 Java 代码很简单，大家一看就懂，main()函数调用 add()函数，add()函数对两个人参进行相加求和并返回累加结果。

执行 `javap -verbose` 命令，分析这段 Java 代码编译成的字节码内容，得到如下输出：

清单：示例 Java 程序所对应的字节码内容

作用：Java 程序所对应的字节码内容分析

```
public static void main(java.lang.String[]);
```

Code:

Stack=2, Locals=1, Args_size=1

0:iconst_5

1:bipush 8

3:invokestatic #2; //Method add:(II)I

6:pop

7:return

LineNumberTable:

line 4: 0

line 5: 7

```
public static int add(int, int);
```

Code:

Stack=2, Locals=4, Args_size=2

0:iload_0

1:iload_1

2:iadd

3:istore_2

4:iload_2

5:bipush 9

7:iadd

8:istore_3

9:iload_3

10:ireturn

LineNumberTable:

line 8: 0

line 9: 4

line 10: 9

```
}
```

这段字节码指令，若你还不能看懂，可以暂且放下，本书后面会对其进行详细讲解。现在我们在 JVM 上运行这个 Java 字节码文件，在运行时打印 JVM 所执行的机器指令（需要为 JVM 安装 HSDIS 插件）。让 JVM 以模板解释器来解释运行这段字节码，JVM 将输出如下字节码与

机器指令的对应内容：

清单：示例 Java 程序

作用：Java 字节码指令所对应的机器指令逻辑

`iload_0 26 iload_0 [0xb370b660, 0xb370b6a0] 64 bytes`

`[Disassembling for mach='i386']`

`0xb370b683: push %eax`

`0xb370b684: mov (%edi), %eax`

`0xb370b686: movzbl 0x1(%esi), %ebx`

`0xb370b68a: inc %esi`

`0xb370b68b: jmp *-0x48ecd6a0(, %ebx, 4)`

`// ...`

`iload_1 27 iload_1 [0xb370b6c0, 0xb370b700] 64 bytes`

`[Disassembling for mach='i386']`

`0xb370b6e3: push %eax`

`0xb370b6e4: mov -0x4(%edi), %eax`

`0xb370b6e7: movzbl 0x1(%esi), %ebx`

`0xb370b6eb: inc %esi`

`0xb370b6ec: jmp *-0x48ecd6a0(, %ebx, 4)`

`// ...`

`iadd 96 iadd [0xb370cf00, 0xb370cf20] 32 bytes`

`[Disassembling for mach='i386']`

`0xb370cf00: pop %eax`

`0xb370cf01: pop %edx`

`0xb370cf02: add %edx, %eax`

`0xb370cf04: movzbl 0x1(%esi), %ebx`

`0xb370cf08: inc %esi`

`0xb370cf09: jmp *-0x48ecd6a0(, %ebx, 4)`

`0xb370cf10: add %al, (%eax)`

`0xb370cf12: add %al, (%eax)`

`// ...`

细心的读者可以发现，这里包含 3 段机器指令，分别对应 JVM 的 3 条字节码指令：`iload_0`、`iload_1` 和 `iadd`。这些信息是 JVM 打印出来的，当 JVM 分别解释执行 `iload_0`、`iload_1` 和 `iadd` 这些字节码指令时，实际上最终执行了其对应的一大串机器码。

实际上 JVM 会将其 200 多个字节码指令所对应的机器码全部打印出来，但是本书限于篇幅，不可能将其全部显示在这里，因此仅挑选 `iload_0`、`iload_1` 和 `iadd` 这 3 条字节码指令。有兴趣的读者可以研究研究这 3 个字节码指令所对应的机器指令，当然本书后面也会详细分析这些指令的含义。在这里我们先不用关心这些机器指令到底代表什么逻辑，只要知道 JVM 在调用 Java

程序时，最终其实执行的是机器指令就可以了。

经过对以上这段示例程序的分析可知，在 JVM 内部一定存在一个“边界”，边界外面是 C 程序，边界里面则直接跳转到机器码。换言之，C 程序可以直接调用汇编指令。

可是即使你翻遍有关 C 语言的教科书，也几乎不会发现有哪本教科书会介绍 C 语言这种能够直接执行机器指令的功能。

似乎是毫无头绪。想当年詹爷也一定在技术路径上有过类似的迷茫和彷徨，不知何去何从（个人瞎掰，若果猜错，请詹爷原谅）。

当然，也许有高手会说，JVM 一开始压根儿不是模板解释器，而是纯粹的字节码解释器，其会将字节码指令逐条翻译成 C 程序。这种说法是完全正确的，但是由于效率实在太低，低到为 C++ 程序员们所不齿，所以 JVM 没发布几个版本就将默认的字节码解释器换成模板解释器了，JVM 执行 Java 函数时直接执行机器指令。但是本书并不会纠结于这些细节，本书主要以启发式的方法来剖析 JVM 内部的运行机制，注重的是主要的思路逻辑，剖析 JVM 为什么会这样做，为什么会使用这样的技术和框架。任何事情都有其必然的道理，如果你对 JVM 能够做到“知其然”，那你已经相当牛了；但是如果你能够对 JVM 做到“知其所以然”，那么你将达到超脱的境界。你可以任意定制自己的 JVM 了，整个底层世界完全由你操控，这是一件想想就激动的事（至少我本人就是这样，当我做了很多次试验终于弄明白了 JVM 的执行引擎后，兴奋得好多天睡不着觉）。当然如果有读者是细节控，那么也请忍一忍啦。

言归正传，所幸的是，C 语言有一个秘密武器，而这种秘密武器就是使 C 语言跨越 IT 界几十年历史长河却依然屹立不倒并一直非常流行的根本和基础，那就是指针。当然，这也是 JVM 得以最终成功的关键因素和关键技术。

可以说，任何底层编程开发，只要祭出“指针”这门神功或者武器，就没有 C 语言搞不定的事（因为能够访问内存，几乎能做机器指令所做的大部分事），所以它一直很流行，各种底层软件、驱动程序、操作系统、关系数据库、办公软件、手机操作系统，等等，几乎都是用它开发而成。同时，相比于汇编，C 语言在语法上更加人性化，更易于被人类接受和理解，因此 C 语言广泛取代了汇编语言，只有在对性能要求非常苛刻的一些领域才需要“祭出”汇编这种最原始、古朴，当然威力也最强大的“神器”，例如视频软件的内存访问或者操作系统底层的原子同步。JVM 内部的原子操作也基本全部采用汇编指令来实现。

那么下面就有请 C 语言的“指针”隆重上场！大部分做 C 程序开发的同学对 C 语言的指针变量都不陌生，例如 `int*`、`char*`，等等，你可以将 `char*` 理解成一个字符串的开始地址，也可以将其理解成一个二维字符数组的首地址。同理，既可以将 `int*` 理解成一个整型变量的内存地址，也可以将其理解成一个二维整型数组的首地址。

而 C 语言的指针其实还有一种更重要的用法——函数指针。通过函数指针，C 语言可以将一个变量直接指向一个函数的首地址。C 语言被编译时，C 函数将被直接编译成机器指令，而这个函数指针将直接指向这段机器指令的首地址。于是聪明的人类便打了个擦边球，在源代码编码阶段就定义好一段机器指令，然后直接将一个 C 函数指针指向这段机器指令的首地址，从而间接实现 C 语言直接调用机器指令的目的（其实，C 语言还有一种办法可以调用汇编指令，那就是内嵌汇编的方式，在 Linux 操作系统内核中就有大量的这种用法。但是这种用法与 JVM 所要实现的目标稍微有点不同，JVM 要实现直接由 C 语言调用机器指令，而内嵌汇编的方式只实现了这一目标的一半，内嵌汇编的方式只能实现由 C 语言直接调用汇编指令。（注意：汇编指令与机器指令之间还有很大差距）。下面来看一个 C 语言直接调用机器指令的示例：

清单：C 程序示例

作用：演示 C 函数直接调用机器指令

//下面这段字符串是一组机器码，使用十六进制表示。这段机器码表示一个函数，能够对传入的参数执行增1操作

//注：下面的 xe9 表示十六进制数，等同于 0xe9

//同时 xe9 前面的斜杠必须是反斜杠\,而不能是正斜杠/

```
const unsigned char code[] =
```

```
"\xe9\x07\x00\x00\x00\xcc\xcc\xcc\xcc\xcc\xcc"
```

"\xcc\x55\x8b\xec\x83\xec\x40\x53\x56\x57\x8d"\

"\x7d\x00\b9\x10\x00\x00\x00\b8\xcc\xcc\xcc"

"\xcc\x33\xab\x8b\x45\x08\x83\xc0\x01\x5f\x5e"

"\x5b\x8b\xe5\x5d\xc3";

```
int main() {
```

```
int result;
```

```
//定义函数指针 fun, 其指向某个函数的入口地址
```

```
int (*fun) (int);
```

```
//初始化函数指针, 将其指向 code 机器码的入口, 这段机器码其实表示一个函数
```

```
fun=(void*) code;
```

```
//调用函数
```

```
result=fun(7);
```

```
printf("result=%d/r/n", result);
```

}

在本示例中，定义了一个全局数组 `code`，`code` 数组里保存的若干字符就是机器指令，这些机器指令能够对 `ra` 执行自增操作。

在 `main()` 函数中通过 `int (*fun)(int)` 定义了一个指针函数 `fun`，接着通过 `fun = (void*)code` 将该指针函数指向一个内存地址，就是 `code` 数组的首地址，最后通过 `result = fun(7)` 就能调用这个

指针函数了。当这段 C 程序被编译后，`fun()` 实际上就指向了 `code` 数组的内存首地址。当物理机器加载这段编译后的程序，当执行到 `result = fun(7)` 这条指令时，就会将 `CS:IP` 段寄存器指向 `code` 首地址，从而将 `code` 数组所在的这一连续内存区域当作代码段来执行。

这就是 JVM 内部能够直接由 C 程序调用和执行机器指令的奥秘所在。其实，这便是本书第 1 章 2.3.3 节所讲的内容，只不过那时并没有直接从 JVM 的角度分析问题。

在 JVM 内部也有这么一个函数指针，就是 `call_stub`。这个函数指针正是 JVM 内部 C 语言程序与机器指令的分水岭，JVM 在调用这个函数之前，主要执行 C 程序（其实还是 C 程序编译后的机器指令），而 JVM 通过这个函数指针调用目标函数之后，就直接进入了机器指令的领域。

`call_stub` 函数指针在 JVM 内部具有举足轻重的意义，在 Java 程序的执行过程中，会有很多地方涉及直接从 JVM 内部调用 Java 函数，例如执行 Java 程序主函数，或者类加载。本章通过介绍函数指针的执行原理，使你更加深入地理解 JVM 跨越 C/C++ 程序而调用 Java 函数的机制。

`call_stub` 函数指针原型定义在 `stubRoutines.hpp` 文件中，定义如下：

清单：/src/share/vm/runtime/stubRoutines.hpp

作用：函数原型定义

```
static CallStub call_stub()
{
    return CAST_TO_FN_PTR(CallStub, _call_stub_entry);
}
```

在这段代码里出现了一个宏：`CAST_TO_FN_PTR`。在 JVM 中，凡是出现函数名大写的情况，基本都是宏。事实上，这基本也是 C/C++ 语言的一种规范。

`CAST_TO_FN_PTR` 宏定义在 `globalDefinitions.hpp` 文件中，定义如下：

```
#define CAST_TO_FN_PTR(func_type, value) ((func_type) (castable_address(value)))
```

将 `call_stub()` 函数体按照这个宏进行替换和展开（C 程序中的宏会在预编译阶段被替换），最终得到如下函数定义：

清单：/src/share/vm/runtime/stubRoutines.hpp

作用：函数原型定义

```
static CallStub call_stub()
{
    return (CallStub) (castable_address(_call_stub_entry));
}
```

现在这种形式又代表了什么含义呢？为了理解这个问题，我们就需要对 C 程序中的函数指针有深入的了解。

不过在讲解之前，先对本节内容做一个简单的总结：Java 字节码指令直接对应一段特定逻辑的本地机器码，而 JVM 在解释执行 Java 字节码指令时，会直接调用字节码指令所对应的本地机器码。JVM 是使用 C/C++ 编写而成的，因此 JVM 要直接执行本地机器码，便意味着必须要能够从 C/C++ 程序中直接进入机器指令。这种技术实现的关键便是使用 C 语言所提供的一种高级功能——函数指针。通过函数指针能够直接由 C 程序触发一段机器指令。

在 JVM 内部，call_stub 便是实现 C 程序调用字节码指令的第一步——例如 Java 主函数的调用。在 JVM 执行 Java 主函数所对应的第一条字节码指令之前，必须经过 call_stub 函数指针进入对应的例程，然后在目标例程中触发对 Java 主函数第一条字节码指令的调用。

2.3 函数指针

在正式解密 call_stub() 之前，有必要讲一下 C 语言中的函数指针概念，只有明白了什么是函数指针，如何调用函数指针，才能真正看懂 call_stub()。函数指针是 C/C++ 语言里一种高级的变量和应用，可能很多做应用开发的 C/C++ 程序员平时也很少接触这种用法，因此大家不妨在这里稍微花点时间看一看。首先需要区分两个概念：函数指针与指针函数。

1. 函数指针与指针函数

在 C 程序中，有两种概念容易混淆，那就是函数指针与指针函数。例如下面定义了一个指针函数：

```
int *fun(int a, int b);
```

而下面则定义了一个函数指针：

```
// 声明一个函数指针 fun
void (*fun)(int a, int b);
```

```
// 声明一个函数原型 add
void add(int x, int y);
```

```
// 为函数指针赋值：将 add() 函数首地址赋值给 fun 指针
fun = add;
```

仔细观察指针函数和函数指针的声明语法，可以发现这两者的一个主要区别：

如果函数名称前面的指针符号 * 没有被括号包含，则所定义的就是指针函数；如果被括号包含，则所定义的就是函数指针。

◎ 在 int *fun(int a, int b) 这行声明中，fun 前面的指针符号 * 没有包含在括号内，所

以这行代码就是在定义一个指针函数。

- ◎ 在 `void (*fun)(int a, int b)` 这行声明中，`fun` 前面的指针符号 `*` 被包含在括号内，所以这行代码就是在声明一个函数指针。

这就是指针函数与函数指针两者之间在形式上的差别。

指针函数与函数指针在形式上只有微小的差别，但是在内容和含义上差别就大了。通俗地讲，指针函数与函数指针在内容上的含义分别如下：

- ◎ 指针函数声明的是一个函数，与一般的函数声明并无多大区别，唯一有区别的就是指针函数的返回类型是一个指针，而一般的函数声明所返回的则是普通变量类型。
- ◎ 函数指针声明的是一个指针，只不过这个指针与一般的指针不同，一般的指针指向一个变量的内存地址，而函数指针则指向一个函数的首地址。

所以，在 `int *fun(int a, int b)` 这行声明中，`fun` 实际上就是一个普通的函数，这个函数包含 2 个人参，类型都是 `int`。同时这个函数返回一个 `int*` 类型的指针。

在 `void (*fun)(int a, int b)` 这行声明中，`fun` 实际上是一个指针变量，注意，是变量，不是函数。这行声明指出，`fun` 指针指向了一个函数，所指向的这个函数必须包含 2 个人参，类型都是 `int`。同时，`fun` 指针所指向的函数必须有一个 `void` 类型的返回值。

下面分别给出指针函数和函数指针的一个示例。

示例一：指针函数

清单：示例程序

作用：C 程序指针函数示例

```
#include<stdio.h>

// 声明指针函数
int *add(int a, int b);

int main(){
    int a = 5;
    int b = 3;

    // 调用指针函数
    int *c=add(a, b);
    printf("c = %p\n", c);

    return 0;
```

```

}

// 定义指针函数
int *add(int a, int b){
    int c=a+b;
    // 这里返回变量 c 的内存地址
    return &c;
}

```

本程序声明了一个指针函数 `add()`，包含两个 `int` 类型的入参，同时返回 `int*` 类型的指针。在 `main()` 主函数调用时，使用与 `add()` 返回值类型相同的指针类型 `int *c` 变量来接收 `add()` 函数所返回的值，并最终打印出返回的指针的值。这个值实际上就是 `add()` 函数内部变量 `c` 的内存地址。打印内容如下：

```
c = 0xbf9932e4
```

示例二：函数指针

清单：示例程序

作用：C 程序函数指针示例

```

#include<stdio.h>

// 声明函数指针 addPointer
int (*addPointer)(int a, int b);

// 声明一个普通函数 add
int add(int a, int b);

int main(){
    int a = 5;
    int b = 3;

    // 初始化函数指针 addPointer，将其指向 add() 函数首地址
    addPointer = add;

    // 调用函数指针所指向的函数
    int c=addPointer(a, b);
    printf("c=%d\n", c);

    return 0;
}

int add(int a, int b){
    int c=a+b;
    return c;
}

```

```
}

```

在本例中，定义了一个函数指针 `addPointer`，并将其指向了函数 `add()`，然后就可以通过 `int c=addPointer(a, b)` 这样的形式，像调用普通函数一样，通过函数指针来调用其所指向的函数。由于 `addPointer` 指针指向了函数 `add()`，因此程序最终实际上调用的就是 `add()` 函数。下面会讲到通过函数指针来调用函数的几种形式。

2. 函数指针的两种定义方式

函数指针通常可以通过两种方式进行声明。

1) 方式一：直接声明

```
return_type (*func_pointer)( data_type arg1,
                             data_type arg2,
                             ...,
                             data_type argn);
```

注意：在定义函数指针时，指针运算符 `*` 一定要使用括号括起来，否则就不是定义函数指针了，而是变成定义指针函数。这一点切记。

使用这种方式定义的函数指针就相当于直接定义了一个变量，可以直接对该变量进行赋值。函数指针与普通变量一样，可以在函数外面声明，作为全局变量，也可以声明在函数内部作为局部变量。例如上面在 `main()` 函数外面所声明的 `addPointer` 这一函数指针，其实也可以声明在 `main()` 函数内部。将上面的例子改成如下：

清单：示例程序

作用：C 程序函数指针示例

```
#include<stdio.h>
```

```
// 声明一个普通函数 add
```

```
int add(int a, int b);
```

```
int main(){
```

```
    int a = 5;
```

```
    int b = 3;
```

```
// 在 main() 函数内部声明函数指针 addPointer
```

```
int (*addPointer)(int a, int b);
```

```
// 初始化函数指针 addPointer，将其指向 add() 函数首地址
```

```
addPointer = add;
```

```
// 调用函数指针所指向的函数
```



```

    int c=addPointer(a, b);
    printf("c=%d\n", c);

    return 0;
}

int add(int a, int b){
    int c=a+b;
    return c;
}

```

在本例中，addPointer 函数指针被声明在了 main() 函数内部，此时它就变成了局部变量。这也是函数指针与指针函数的区别之一，指针函数毕竟本质上仍然是一个函数，而函数是不可以被声明在其他函数内部的。

2) 方式二：通过类型声明

函数指针还有一种声明的方式，那就是先定义一个类型，然后通过所定义的类型去声明函数指针。

```

typedef (*func_pointer)( data_type arg1,
                        data_type arg2,
                        ...,
                        data_type argn);

```

注意：这里不是在声明函数指针，而是定义了一种函数指针的类型。这种类型的类型名就是 func_pointer。

类型声明的方式和直接声明的方式相比，主要区别在于，通过类型声明的方式定义的仅仅是一个类型，由这个类型无法直接去初始化函数指针，因为类型不是变量。例如，在这里定义了 func_pointer 这个类型后，并不能对其进行初始化并将其指向某个函数。要想使用这种类型，就必须使用类型名去声明一个函数指针变量，然后才能为所声明的函数指针变量赋值。

还是拿刚才的 addPointer 举例，说明如何通过类型定义的方式来声明函数指针：

清单：示例程序

作用：C 程序函数指针示例

```

#include<stdio.h>

// 声明一个普通函数 add
int add(int a, int b);

int main(){
    int a = 5;
    int b = 3;
}

```

```

// 定义 addPointerType 这一类型
typedef (*addPointerType)(int a, int b);

// 声明一个 addPointerType 类型的变量
addPointerType addPointer;

// 初始化函数指针 addPointer, 将其指向 add() 函数首地址
addPointer = add;

// 调用函数指针所指向的函数
int c=addPointer(a, b);
printf("c=%d\n", c);

return 0;
}

int add(int a, int b){
    int c=a+b;
    return c;
}

```

在本例中, 先通过 `typedef (*addPointerType)(int a, int b)` 定义了一种类型, 然后再声明了一个变量 `addPointer`, 变量 `addPointer` 就属于 `addPointerType` 这种类型。其实, `addPointerType` 就类似于 C 程序中的 `int`、`char` 等基本类型, 只不过 `int`、`char` 等基本类型是内建的, 是 C 程序本来就支持的类型, 而 `addPointerType` 则是开发者自定义的一种类型。

如果使用 Java 程序来举例, `addPointerType` 好比是 Java 开发者自定义的一个 Java 类, 定义好 Java 类后, 就可以声明属于这种 Java 类型的变量, 并对变量进行初始化。

在 C 语言中, 通过类型定义的方式来声明函数指针, 是一种比较普遍的做法。JVM 中的 `call_stub` 函数指针便是通过这种方式来声明的。下面对该函数指针进行详细的剖析。

了解了函数指针的声明方式后, 还需要了解函数指针的常用方式。

3. 函数指针的两种调用方式

函数指针有两种基本调用方式。

假设已经声明并定义一个函数指针 `funcPointer`, 则可以通过如下两种格式来调用:

- ◎ `(*funcPointer)(参数列表)`
- ◎ `funcPointer(参数列表)`

第二种格式, 看起来好像 `funcPointer` 就是一个普通的函数名, 如果不看其定义方式, 根本

看不出来这到底是一个指针，还是一个函数。

第一种格式看起来比较古怪，但是有相当一部分人喜欢这么使用。之所以使用这种格式，是因为这种格式可以让人知道这是在通过指针而非函数进行函数调用。

还以上面的 `addPointer` 函数指针为例，下面分别给出这个指针的两种调用方式。在上面的例子中，通过第二种格式进行调用，如下：

清单：示例程序

作用：C 程序函数指针示例

```
int add(int a, int b);

int main(){
    //...

    int (*addPointer)(int a, int b);
    addPointer = add;

    // 调用函数指针所指向的函数
    int c=addPointer(a, b);
    //...
}
```

本例通过 `addPointer(a, b)` 这种格式调用函数指针。

清单：示例程序

作用：C 程序函数指针示例

```
int add(int a, int b);

int main(){
    //...

    int (*addPointer)(int a, int b);
    addPointer = add;

    // 调用函数指针所指向的函数
    int c=(*addPointer)(a, b);
    //.....
}
```

本例通过 `(*addPointer)(a, b)` 这种格式调用函数指针。

在 JVM 中，我们即将分析的 `call_stub` 函数指针便通过这种格式进行调用。

总体而言，函数指针作为 C 语言中的高级应用，是实现 C 语言动态扩展能力的关键技术之一，如同 Java 中的反射与类动态加载技术。

函数指针通常有两种定义方式，一种是像定义普通变量一样定义，一种是通过 `typedef` 关键字进行类型声明。

函数指针本质上是一种指针，不是函数，但是由于这种指针指向的并不是某个变量的内存地址，而是某个函数的内存首地址，因此 C 语言允许像调用函数一样调用函数指针。要注意的是，普通的指针是不能被当成函数去调用的，这便是函数指针的奇特之处——与普通的指针变量在概念上完全一致但是能够被当成函数进行调用，而与函数在调用上具有相同的方式但是在概念上却有巨大的差异。

函数指针通常有两种调用方法，在 JVM 内部，使用了其中一种比较奇特的调用方式。如果不了解这种奇特的调用方式，很难理解 JVM 内部的实现机制。

2.4 CallStub 函数指针定义

上一节介绍了 C 语言中的函数指针与指针函数的概念、区别、声明方式及调用格式。现在再次将目光聚焦到 JVM 的 `call_stub` 指针上，这种指针其实就是函数指针。前面讲过，`call_stub` 函数指针在 JVM 内部具有举足轻重的作用，例如 Java 程序主函数的调用链路就必须经过 `call_stub` 函数指针的执行，本书后面章节在讲解类加载机制时会提到，Java 类的加载链路也会经过该函数指针。因此，对于 JVM 执行引擎而言，该函数指针无比重要，没有这个指针的存在，JVM 执行引擎便无从谈起。因此本章将会详细讲解 `call_stub` 的调用机制。

上面讲过，将下面这句代码：

```
return CAST_TO_FN_PTR(CallStub, _call_stub_entry);
```

进行宏替换后，得到下面这行展开式：

```
return (CallStub)(castable_address(_call_stub_entry));
```

这里的 `CallStub` 其实就是一种自定义的类型。先不用管 `CallStub` 究竟是怎样的一种类型，为了将问题简化，可以将其想象成最简单的一种类型，例如 `int`，使用 `int` 这种基本类型替换 `CallStub`，就得到下面的替换式：

```
return (int)(castable_address(_call_stub_entry));
```

这么一替换，这种形式立刻就变成司空见惯的形式，`castable_address(_call_stub_entry)` 返回了一个结果类型，JVM 又将这种类型转换成了 `int` 类型。

现在一起看看 `CallStub` 究竟是怎样的一种类型。`CallStub` 定义在 `/src/share/vm/runtime/stubRoutines.hpp` 文件中，声明如下：

清单：/src/share/vm/runtime/stubRoutines.hpp

作用：类型定义

```
// Calls to Java
typedef void (*CallStub) (
    address link,
    intptr_t* result,
    BasicType result_type,
    methodOopDesc* method,
    address entry_point,
    intptr_t* parameters,
    int size_of_parameters,
    TRAPS
);
```

由该定义可知，CallStub 是这样的一种函数指针类型：其指向的函数，返回值类型是 void，并且有 8 个入参。

到这里，call_stub() 函数调用的方式基本理顺了。但是，前文在讲解函数指针时曾提到函数指针的两种调用方式，而在 call_stub() 里似乎没有看到任何一种格式的调用。我们先看看 JVM 内部是如何调用 call_stub() 的。JVM 在 javaCalls::call_helper() 中执行了 call_stub() 函数调用，其调用的源代码如下：

清单：/src/share/vm/runtime/javaCalls.cpp::call_helper()

作用：演示 JVM 内部调用 call_stub()

```
void JavaCalls::call_helper(JavaValue* result,
    methodHandle* m,
    JavaCallArguments* args,
    TRAPS) {
    //...
    // call_stub() 调用开始
    // do call
    { JavaCallWrapper link(method, receiver, result, CHECK);
        { HandleMark hm(thread); // HandleMark used by HandleMarkCleaner

            StubRoutines::call_stub() (
                (address)&link,
                // (intptr_t*)&(result->_value),
                result_val_address,
                result_type,
                method(),
                entry_point,
                args->parameters(),
                args->size_of_parameters(),
```

```

        CHECK
    );

    // circumvent MS C++ 5.0 compiler bug (result is clobbered across call)
    result = link.result();
    // Preserve oop return value across possible gc points
    if (oop_result_flag) {
        thread->set_vm_result((oop) result->get_jobject());
    }
}
}
// call_stub() 调用结束

//...
}

```

JVM 内部直接调用了 `call_stub()`，并且传入了 8 个参数，可是 `call_stub()` 函数原型却是没有人参的（如下所示，其实上文已经贴过一次了，不过这里为了阐述问题，再贴一次）。这又是怎么回事呢？

```

static CallStub call_stub()
{
    return (CallStub)(castable_address(_call_stub_entry));
}

```

（注意：在 `call_stub()` 的函数原型声明里，并没有人参，而 JVM 在调用 `call_stub()` 函数时却传入了 8 个参数。）

其实，这里 JVM 隐式地调用了函数指针。`call_stub()` 函数最终返回的是一个函数指针的实例变量（C 语言中没有实例的概念，这里借用下 Java 的实例概念，意在强调这是一个初始化了的变量）。虽然 `call_stub()` 的原型函数里只有 `return (CallStub)(castable_address(_call_stub_entry))` 这一行代码，可是这行代码所蕴含的逻辑却十分丰富，编译器编译后，这行代码会被转换为类似下面的形式：

清单：/src/share/vm/runtime/stubRoutines.hpp

作用：函数原型定义

```

static CallStub call_stub()
{
    // 使用 CallStub 类型声明一个函数指针变量
    CallStub functionPointer;

    // 调用 castable_address(), 并用 address 类型的变量 returnAddress 接收返回值
    // castable_address() 函数返回的变量类型就是 address，这个后面马上就会讲到
    address returnAddress = castable_address(_call_stub_entry);
}

```

```

// 将 address 类型的变量转换为 CallStub 类型的变量
functionPointer = (CallStub) returnAddress;

// 返回 CallStub 类型的变量
return functionPointer;
}

```

高级语言之所以高级，就是人类往往能够只使用简短的几行代码（甚至一行）就能表达出十分丰富的含义，一切含义的表达皆由编译器在幕后默默支撑。当然，编译器并不会真的将 C 源程序转换成上面这段代码，只不过编译器会自动生成中间变量，最终所表达出来的逻辑就与上面这段代码相同。

在这段代码里，我们发现，其实在 C 程序里通过 `typedef` 所自定义的类型，也可以参与类型强制转换的计算。在本例中，`castable_address(_call_stub_entry)` 返回的其实是 `address` 这种自定义的类型，而编译器最终将其转换成了 `CallStub` 这种自定义的类型。

`call_stub()` 最终返回一个 `CallStub` 类型的函数指针变量，调用者其实就可以像调用普通函数那样来调用这种函数指针变量。只不过 JVM 并没有显式地调用函数指针，而是隐式地进行了调用（即偷偷摸摸、不声不响地，`--`）。但是物理机器是不会理解这种隐式调用的，最终还是要靠编译器来实现隐式调用到显式调用的转变。编译器处理后的显式调用可以用下面这段逻辑表达：

清单：/src/share/vm/runtime/javaCalls.cpp::call_helper()

作用：演示 JVM 内部 `call_stub()` 的显式调用

```

void JavaCalls::call_helper(JavaValue* result,
                             methodHandle* m,
                             JavaCallArguments* args,
                             TRAPS) {

    //...

    // call_stub() 调用开始
    // do call
    { JavaCallWrapper link(method, receiver, result, CHECK);
      { HandleMark hm(thread); // HandleMark used by HandleMarkCleaner

        // 先声明一个 CallStub 类型的函数指针变量
        CallStub funcPointer;

        // 初始化函数指针变量，将其指向 call_stub() 函数首地址
        funcPointer = StubRoutines::call_stub();

        // 调用函数指针变量，传入 8 个参数
        funcPointer(

```



```

        (address)&link,
        // (intptr_t*)&(result->_value),
        result_val_address,
        result_type,
        method(),
        entry_point,
        args->parameters(),
        args->size_of_parameters(),
        CHECK);

    // circumvent MS C++ 5.0 compiler bug (result is clobbered across call)
    result = link.result();
    // Preserve oop return value across possible gc points
    if (oop_result_flag) {
        thread->set_vm_result((oop) result->get_jobject());
    }
}
// call_stub() 调用结束
//...
}

```

这样一分解，JVM 实现 `call_stub()` 的机制就非常清晰了（其实现在再说调用 `call_stub()`，大家都知道这种说法是错误的，因为实际上 JVM 调用的是 `call_stub()` 函数所返回的函数指针变量），相信一般人都能看得很明白。由于 JVM 在使用 `typedef` 定义 `CallStub` 类型时，就规定这种函数指针类型有 8 个入参，因此 JVM 最终在调用这种类型的函数指针时，也必须传入 8 个类型完全相同的参数。

1. `castalbe_address()`

上文在讲解函数指针的声明时谈到，定义了一种函数指针的类型后，可以去声明属于这种类型的一个函数指针变量，并可以初始化这个变量，使其指向某个函数。

注意：上面对函数指针的初始化，都是将其指向某个函数，而实际上，由于函数指针也是一种指针（如同海马也是马），而指针的特点就是可以指向内存的任意地址，既可以指向函数的首地址，也可以指向某个变量的首地址，说白了，其实指针里无非就是存储了一个值而已。所以，函数指针也是既能指向函数首地址，也能指向某个变量首地址，或者指向任何你想指向的内存地址。

而 `call_stub()` 函数内部其实就是让函数指针指向了某个内存地址。

call_stub()函数宏展开后的逻辑是：

```
return (CallStub)(castable_address(_call_stub_entry));
```

这里的 castable_address 是一个函数，定义在 globalDefinitions.hpp 文件中，其定义如下：

```
inline address_word castable_address(address x)
{
    return address_word(x);
}
```

这里的 address_word 也是一种自定义类型，顾名思义，它表示的是一种地址类型。该类型在 globalDefinitions.hpp 中定义，定义如下：

```
typedef uintptr_t address_word;
```

由此可知，address_word 类型其实是 uintptr_t，而后一种类型也是 JVM 自定义的类型。但是这种类型是平台相关的，所以在 JVM 内部有 3 处定义了这种类型，分别是：

- ◎ globalDefinitions_gcc.hpp
- ◎ globalDefinitions_sparcWorks.hpp
- ◎ globalDefinitions_visCPP.hpp

在特定的平台上编译 JVM 时，编译器会自动根据平台类型，编译不同的 hpp 头文件。这里所列出的 3 种头文件，只看名字就能知道，分别对应的是 Linux、Macintosh 和 Windows 这 3 种主流的操作系统。这里之所以列出这 3 种文件，并不是想让大家去研究一番，而是让大家感受一下，如果使用 C/C++ 语言编程，程序员必须要考虑平台架构的异构性，并且必须在代码中处理平台的兼容性。当年詹爷也是受够了这种麻烦，所以才会搞出个 Java 跨平台的编程语言出来。所以，Java 程序员是幸福的。

在 Linux 平台上，uintptr_t 类型的定义如下（globalDefinitions_gcc.hpp 文件中）：

```
typedef unsigned int uintptr_t;
```

根据这个定义可知，uintptr_t 在 Linux 平台上的类型原型是 unsigned int，这是 C 语言的基本类型之一，即无符号整数类型。于是，绕了一小圈，终于得知 address_word 这种类型的“庐山真面目”。

让我们再把目光转回到 call_stub()函数，将 address_word 类型进行替换，得到如下代码：

```
static CallStub call_stub()
{
    // 下面的代码原本是这一句：return (CallStub)(castable_address(_call_stub_entry));
    return (CallStub)(unsigned int(_call_stub_entry));
}
```

到了这一步，`call_stub()`函数的逻辑基本全部还原出来了，`call_stub()`函数的逻辑总结起来包含下面两步：

- ◎ 第一步，将 `_call_stub_entry` 变量转换为 `unsigned int` 这一基本类型。
- ◎ 第二步，将 `_call_stub_entry` 所转换后的 `unsigned int` 这一基本类型再转换为 `CallStub` 这一自定义类型，该类型是函数指针类型。

那么 `_call_stub_entry` 是啥？是啥类型？该类型定义在 `StubRoutines.hpp` 中，定义如下：

```
static address _call_stub_entry;
```

由此可知，`_call_stub_entry` 本身就是 `address` 类型，而该类型的原型是 `unsigned int`。

再次回顾下 JVM 调用 Java 函数的过程：

- ◎ JVM 先调用 `call_stub()`函数，该函数将 `_call_stub_entry` 这一 `unsigned int` 类型的变量转换成 `CallStub` 自定义的类型，该类型是函数指针。
- ◎ JVM 将 `call_stub()`所返回的函数指针当成函数进行调用。

在这个过程中，似乎还有件事不明确，那就是函数指针的指向。在 `call_stub()`里是直接返回了 `_call_stub_entry` 这一基本类型的变量，然后直接就被转换成了 `CallStub` 这一自定义的函数指针类型，接着 JVM 直接就调用该函数指针了，自始至终并没有看到函数指针被指向了哪个函数。而前文讲解函数指针时提到，要调用函数指针，必须将其指向某个函数。

其实，JVM 在初始化的过程中，便将 `_call_stub_entry` 这一变量指向了某个内存地址，否则 JVM 肯定无法直接调用。在 x86 32 位 Linux 平台上，JVM 在初始化过程中，存在这样一条链路：

```
java.c: main()
  java_md.c: LoadJavaVM()
  jni.c: JNI_CreateJavaVM()
    Threads.c: create_vm()
      init.c: init_globals()
        StubRoutines.cpp: stubRoutines_init1()
          StubRoutines.cpp: initialize1()
            stubGenerator_x86_x32.cpp: StubGenerator_generate()
              stubGenerator_x86_x32.cpp: StubCodeGenerator()
                stubGenerator_x86_x32.cpp: generate_initial()
```

这条链路从 JVM 的 `main()`函数开始，调用到 `init_globals()`这个全局数据初始化模块，最后再调用到 `StubRoutines` 这个例程生成模块，最终在 `stubGenerator_x86_32.cpp: generate_initial()` 函数中会执行如下代码，对 `_call_stub_entry` 这个变量进行初始化，如下所示：

清单：/src/share/vm/runtime/stubGenerator_x86_x32.cpp

作用：函数原型定义

```
void generate_initial() {
    // Generates all stubs and initializes the entry points

    StubRoutines::_call_stub_entry =
        generate_call_stub(StubRoutines::_call_stub_return_address);

    // ...
}
```

最终, JVM 调用 `generate_call_stub(StubRoutines::_call_stub_return_address)` 函数返回一个值, 赋值给 `_call_stub_entry`。 `generate_call_stub(StubRoutines::_call_stub_return_address)` 里面的逻辑十分重要, 可以说是 JVM 最核心的功能。不过要分析清楚这个最核心的功能模块, 得先弄清楚 `call_stub()` 的入参。

2. CallStub()入参

准确地说, `CallStub` 并不是函数, 因此本节标题写为“`CallStub()`入参”不够严谨, 但是 `CallStub` 是一个函数指针, 最终 JVM 也是通过这一函数指针调用其所指向的函数的, 所以将其说成是函数也不为过。为简单起见, 我们统一说成是“`CallStub()`入参”。

在 `javaCalls.cpp::call_helper()` 函数中, JVM 是这样调用 `CallStub()` 函数的:

清单：/src/share/vm/runtime/javaCalls.cpp::call_helper()

作用：演示 JVM 内部调用 `call_stub()`

```
StubRoutines::_call_stub() (
    (address)&link,
    // (intptr_t*)&(result->_value),
    result_val_address,
    result_type,
    method(),
    entry_point,
    args->parameters(),
    args->size_of_parameters(),
    CHECK
);
```

(注：为节省篇幅, 不再贴出这段代码的上下文。)

JVM 一共传入了 8 个参数, 这 8 个参数的含义如表 2.1 所示。

表 2.1 8 个参数说明

参 数 名	含 义
link	顾名思义，这是一个“连接器”，注意，不是“链接器”
result_val_address	函数返回值地址
result_type	函数返回类型
method()	JVM 内部所表示的 Java 方法对象
entry_point	JVM 调用 Java 方法的例程入口。JVM 内部的每一段例程都是在 JVM 启动过程中预先生成好的一段机器指令。要调用 Java 方法，都必须经过本例程，即需要先执行这段机器指令，然后才能跳转到 Java 方法字节码所对应的机器指令去执行
parameters()	Java 方法的入参集合
size_of_parameters()	Java 方法的入参数量
CHECK	当前线程对象

这些参数每一个都非常重要，下面先进行一个简单介绍。要想深入了解这些参数背后的意义，必须要等到研究完 Java 的内存模型之后。本书在讲解 Java 的内存模型时，会对这些入参进行深入分析。

这里先简单介绍 5 个参数:link、method()、entry_point、parameters()和 size_of_parameters()。

1) 连接器 link

连接器 link 的作用，从其名称也可猜测一二，就是起到连接、桥梁的作用。连接谁呢？这要从 link 的类型定义来说。连接器 link 所属类型是 JavaCallWrapper，该类型定义在 javaCalls.cpp 文件中，定义如下：

```
清单：/src/share/vm/runtime/javaCalls.cpp::JavaCallWrapper
作用：JavaCallWrapper 类定义
// A JavaCallWrapper is constructed before each JavaCall and destructed after
the call.
// Its purpose is to allocate/deallocate a new handle block and to save/restore
the last
// Java fp/sp. A pointer to the JavaCallWrapper is stored on the stack.

class JavaCallWrapper: StackObj {
    friend class VMStructs;
private:
    JavaThread*      _thread;          // the thread to which this call belongs
    JNIHandleBlock*  _handles;         // the saved handle block
    methodOop        _callee_method;  // to be able to collect arguments if
```

```

entry frame is top frame
    oop      _receiver; // the receiver of the call (if a non-static call)

    JavaFrameAnchor _anchor; // last thread anchor state that we must restore

    JavaValue*      _result; // result value

};

```

这是一个 C++ 类型，这个类里面包含这样几个变量：

- ◎ `_thread`，当前 Java 函数所在线程
- ◎ `_handles`，本地调用句柄
- ◎ `_callee_method`，调用者方法对象
- ◎ `_receiver`，被调用者（非静态 Java 方法）
- ◎ `_anchor`，Java 线程堆栈对象
- ◎ `_result`，Java 方法所返回的值

通过这些变量可知，link 其实在 Java 函数的调用者与被调用者之间搭建了一座桥梁，通过这座桥梁，我们可以实现堆栈追踪，可以得到整个方法的调用链路。

在 Java 函数调用时，link 指针将被保存到当前方法的堆栈中。

注意：JVM 内部有一个 linker，是链接器，与 link 是不同的两个对象。

2) method()

method() 是当前 Java 方法在 JVM 内部的表示对象。

每一个 Java 方法在被 JVM 加载时，JVM 都会在内部为这个 Java 方法建立函数模型，说白了就是保存一份 Java 方法的全部原始描述信息。JVM 为 Java 方法所建立的模型中至少包含以下信息：

- ◎ Java 函数的名称、所属的 Java 类
- ◎ Java 函数的入参信息，包括入参类型、入参参数名、入参数量、入参顺序等
- ◎ Java 函数编译后的字节码信息，包括对应的字节码指令、所占用的总字节数等
- ◎ Java 函数的注解信息
- ◎ Java 函数的继承信息
- ◎ Java 函数的返回信息

JVM 在调用 CallStub() 函数指针时，将 method() 对象传递进去，最终就是为了从 method() 对象中获取到 Java 函数编译后的字节码信息，JVM 拿到字节码信息之后，就能对字节码进行解

释执行了。

注：大家通过 Java 反射对象不仅能够获取一个 Java 类的元信息，也能够获取 Java 类中函数的全部原始信息，之所以能够获取，就是因为 JVM 在内部为每一个 Java 类、每一个 Java 方法都建立了内存模型，保存 Java 类和 Java 方法的全部信息，因此在运行期通过反射只需访问这个内存模型就能得到这些信息。这是 Java 语言与 C++（两种都是面向对象的编程语言）的最大不同。C++ 程序由于编译后直接变成了二进制机器指令，已经擦除了所有的面向对象信息，因此 C++ 程序在运行期是获取不到 C++ 类和函数的原始信息的（当然著名的 RTTI 能够为 C++ 提供类型反射的能力）。

3) entry_point

entry_point 是继 `_call_stub_entry` 这一 JVM 最核心例程之后的又一个最主要的例程入口。前面对 `_call_stub_entry` 进行了简单的分析，JVM 每次从 JVM 内部调用 Java 函数时（相对于通过字节码指令调用目标 Java 函数），必定调用 `CallStub` 函数指针，而该函数指针的值就是 `_call_stub_entry`。JVM 通过 `_call_stub_entry` 所指向的函数地址，最终调用到 Java 函数。

在 JVM 通过 `_call_stub_entry` 所指向的函数调用 Java 函数之前，必须要先经过 `entry_point` 例程。事实上，在 `entry_point` 例程里面会真正从 `method()`（刚刚讲过，这是 Java 函数在 JVM 内部的模型）对象上拿到 Java 函数编译后的字节码，JVM 通过 `entry_point` 可以得到 Java 函数所对应的第一个字节码指令，然后开始调用 Java 函数。

这里先简单绘制一下 JVM 经过 `_call_stub_entry`，到 `entry_point`，再到 Java 程序 `main()` 主函数的路线图，如图 2-12 所示。

对于该图大家一定充满了疑惑，一方面是函数指针的调用方式实在比较特殊，如果没有这方面的开发经验，简直有点颠覆大家对函数调用的一般性认识；另一方面，`_call_stub_entry` 和 `entry_point` 这两个例程的概念也实在是云里雾里。现在我只能说，这两个例程其实就是两段机器指令，JVM 提前写好了很多例程，例如函数进入、函数调用、函数返回、异常处理、静态函数调用、本地函数调用，等等，这些例程都是直接使用机器指令写成。直接使用机器指令编写而成的这些例程，能够最大程度地提高程序运行的效率。当然，JVM 中所定义的 200 多种字节码指令，每一条字节码指令也有一个例程，也全部直接使用机器指令写成。

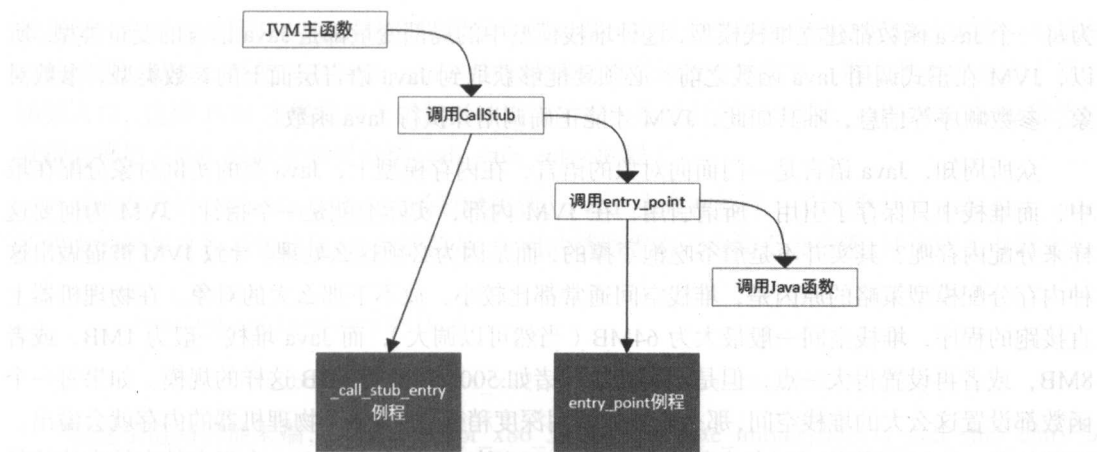


图 2.12 JVM 调用 Java 程序 main() 主函数的路线图

不过任何复杂的事，其实都是可以用简单的道理说明白的，如果说 JVM 中的所谓例程的概念比较玄乎，那么对于广大 Java 程序员来说，有一个名词一定是耳熟能详的，那就是——工具类。我们就将例程想象成一个个工具类即可，个人认为这两者之间完全具有可类比性。从所要实现的功能来看，无论是例程，还是工具类，最终的目的无非就是实现某一个特定的逻辑而已。两者所不同的只是，例程是直接机器指令写的，而 Java 工具类则是使用 Java 语言写的。（严格来说，例程也不是直接用机器指令写的，而是用 C 语言在运行期动态生成的。不过本书意在讲述 JVM 主要的实现思路，以及为什么会选择各种奇妙和深奥的技术，所以如果过分注重细节，一方面比较绕，另一方面也容易“歪楼”，偏离主题，误导思路。所以本书对太细节的东西不会讲得那么清楚，望大家理解。当然，不注重细节，不代表不严谨。）

`_call_stub_entry` 和 `entry_point` 这两个例程的讲解到此先告一段落，后面会深入讲解其机器指令，对其逻辑一探究竟。我们还是继续回到主题，分析 `CallStub()` 的入参。

4) parameters()

这个参数看名字就能猜出其含义，事实上也的确如你所猜，这个参数就是描述 Java 函数的入参信息的。

在 JVM 真正调用 Java 函数的第一个字节码指令之前，JVM 会在 `CallStub()` 函数中解析 Java 函数的入参，解析后，JVM 会为 Java 函数分配堆栈，并将 Java 函数的入参逐个入栈。这样，JVM 就为高层次的编程语言建立了方法栈模型。

相比于 C/C++/Delphi 等编译型语言，Java 这门解释型语言最大的不同就是，不直接在物理机器上运行，而是运行在虚拟机上，所以不能直接使用物理机器的方法栈，必须在虚拟机层面

为每一个 Java 函数都建立堆栈模型，这种堆栈模型中的局部变量都是 Java 语言的变量类型。所以，JVM 在正式调用 Java 函数之前，必须要能够获取到 Java 语言层面上的参数类型、参数对象、参数顺序等信息，唯其如此，JVM 才能正确调用并执行 Java 函数。

众所周知，Java 语言是一门面向对象的语言，在内存模型上，Java 类的实例对象分配在堆中，而堆栈中只保存了引用。所谓引用，在 JVM 内部，实际上就是一个指针。JVM 为何要这样来分配内存呢？其实并不是詹爷吃饱了撑的，而是因为必须这么处理。导致 JVM 被逼做出这种内存分配模型策略的原因是，堆栈空间通常都比较小，放不下那么大的对象。在物理机器上直接跑的程序，堆栈空间一般最大为 64MB（当然可以调大），而 Java 堆栈一般为 1MB，或者 8MB，或者再设置得大一点，但是不可能达到诸如 500MB 甚至 1GB 这样的规模。如果每一个函数都设置这么大的堆栈空间，那么函数的调用深度稍微一大，整个物理机器的内存就会溢出。正因如此，JVM 只能将类对象实例保存到堆中。否则，如果我们定义一个很大的字符串传给某个函数，如果整个字符串信息都保存在堆栈中，那么 JVM 很容易就会被各种大字符串攻击。

关于 Java 堆栈模型以及对象引用模型就先介绍到这里，后文会进行详细深入的分析。

5) size_of_parameters()

这个入参也是见其名就知其意，其含义就是参数数量，即 Java 函数的入参个数。

刚才讲到另一个入参，parameters()，这个参数保存了 Java 函数的所有入参信息，但是 parameters() 里面其实是使用指针建立起来的数组模型，JVM 在后面调用 Java 函数时，直接通过 parameters() 内部的指针，无法得知结束位置，因此这里需要将 Java 函数的入参数量传递进去，JVM 在为 Java 函数分配堆栈空间时，会根据这个值，计算出 Java 堆栈空间大小。

总体而言，CallStub 例程是 JVM 内部举足轻重的一个功能，而 JVM 在调用 CallStub 函数指针时，可谓是层层包装，让人一眼看不出究竟。本节通过抽丝剥茧和场景还原，剖析了 CallStub 这种函数指针的调用机制，描述了 CallStub 函数指针的定义、调用和入参。

2.5 _call_stub_entry 例程

上面花费不少精力讲完 CallStub() 入参，接下来就可以分析 _call_stub_entry 这个例程所指向的一段机器指令的逻辑了。由于这段机器指令牵扯的东西太多（例如汇编基础、Java 函数入参、Java 堆栈模型等），因此前面才会花那么多篇幅进行相关技术的介绍。

前面在讲 JVM 调用 CallStub() 时，提到 JVM 先调用 call_stub() 函数，返回 _call_stub_entry，然后将 _call_stub_entry 强制转换为 CallStub 这种自定义的函数指针类型，最终 JVM 调用这一函

函数指针，实现由 C 程序的世界转入机器指令的世界，完成分水岭的跨越。不过这里有一个最核心的问题，就是，既然 `_call_stub_entry` 最终被转化成了函数指针类型，那么其必定指向了某个函数入口，这样 JVM 才能将这个函数指针当成函数一样进行调用。那么 `_call_stub_entry` 最终究竟指向哪里了呢？这就有必要分析 `_call_stub_entry` 例程了。

前面在分析 `castable_address()` 函数时提到，JVM 中存在这样一条链路，对 `_call_stub_entry` 所代表的例程进行了初始化，这条链路是（下面的链路省略了中间多个步骤，详情请参考前文）：

```
java.c: main()
  java_md.c: LoadJavaVM()
  // ...
  stubGenerator_x86_x32.cpp: generate_initial()
```

在这条链路的最末端，`stubGenerator_x86_32.cpp: generate_initial()` 函数对 `_call_stub_entry` 变量进行了初始化，如下所示：

```
StubRoutines::_call_stub_entry =
  generate_call_stub(StubRoutines::_call_stub_return_address);
```

`generate_call_stub()` 函数返回值赋给了 `_call_stub_entry` 变量，`generate_call_stub()` 函数顾名思义，就是产生 `_call_stub_entry` 变量所指向的一个函数首地址。`generate_call_stub()` 函数的定义如下：

清单：/src/cpu/x86/vm/stubGenerator_x86_32.cpp

作用：generate_call_stub() 函数

```
address generate_call_stub(address& return_address) {
    StubCodeMark mark(this, "StubRoutines", "call_stub");
    address start = __ pc(); // 这里先得到当前入口的内存地址，因为本函数后续流程会
    继续往代码空间中写入 call_stub() 的指令，而外面却只需要拿到这部分指令的首地址，而这里的
    start 就是首地址，因此这里先拿到，啥也不干，最后直接将其返回出去即可

    // stub code parameters / addresses
    assert(frame::entry_frame_call_wrapper_offset == 2, "adjust this code");
    bool sse_save = false;
    const Address rsp_after_call(rbp, -4 * wordSize); // same as in
    generate_catch_exception()!
    const int locals_count_in_bytes (4*wordSize);
    const Address mxcsr_save (rbp, -4 * wordSize);
    const Address saved_rbx (rbp, -3 * wordSize);
    const Address saved_rsi (rbp, -2 * wordSize);
    const Address saved_rdi (rbp, -1 * wordSize);
    const Address result (rbp, 3 * wordSize);
    const Address result_type (rbp, 4 * wordSize);
    const Address method (rbp, 5 * wordSize);
    const Address entry_point (rbp, 6 * wordSize);
    const Address parameters (rbp, 7 * wordSize);
```

```

const Address parameter_size(rbp, 8 * wordSize);
const Address thread      (rbp, 9 * wordSize); // same as in
generate_catch_exception()!
sse_save = UseSSE > 0;

// stub code
__ enter();//以 x86 为例, 在 assembler_x86.cpp 中, enter() 函数实现是:
push(rbp); mov(rbp, rsp);而这两句代码都调用 emit(), 向代码空间中写入机器码
movptr(rcx, parameter_size); //注意: 这些看似汇编指令的 C 函数并不是在运行
时被真正调用的函数, 它们的作用只不过是往代码空间中写入具有相同作用的机器码
__ shlptr(rcx, Interpreter::logStackElementSize); // convert parameter
count to bytes
__ addptr(rcx, locals_count_in_bytes);           // reserve space for
register saves
__ subptr(rsp, rcx);
__ andptr(rsp, -(StackAlignmentInBytes)); // Align stack

// save rdi, rsi, & rbx, according to C calling conventions
__ movptr(saved_rdi, rdi);
__ movptr(saved_rsi, rsi);
__ movptr(saved_rbx, rbx);
// save and initialize %mxcsr
//...

__ BIND(is_double);
// interpreter uses xmm0 for return values
if (UseSSE >= 2) {
    __ movdbl(Address(rdi, 0), xmm0);
} else {
    __ fstp_d(Address(rdi, 0));
}
__ jmp(exit);

return start;
}

```

这段代码最主要的作用就是生成机器码,使用 C 语言动态生成。弄懂这段机器指令的逻辑,对理解 JVM 的字节码执行引擎至关重要。下面我们对这段代码进行详细分析。

1. pc()函数

首先看第一行代码:

```
address start = __ pc();
```

这行代码保存当前例程所对应的一段机器码的起始位置。pc()函数定义如下:

清单：/src/share/vm/asm/assembler.hpp

作用：pc()函数定义

```
address pc() const {
    return _code_pos;
}
```

该函数特别简单，返回_code_pos 变量。该变量也定义在/src/share/vm/asm/assembler.hpp 文件中，是一个 address 类型的变量。

在 JVM 启动过程中，JVM 会生成很多例程（即一段固定的机器指令，能够实现一种特定的功能逻辑），例如函数调用、字节码例程、异常处理、函数返回等。

每一个例程，一开始都有这么一行代码（即 address start = __ pc()），代码完全相同。事实上，JVM 的所有例程都在一段连续的内存中，我们可以将这段内存想象成一根直线，当 JVM 刚启动时，这根线长度为 0，没有生成任何例程。第一个例程生成时，__ pc()返回 0，因为此时是从这根直线的零点位置开始。假设第一个例程占 20 字节，则当 JVM 生成第二个例程时，第二个例程执行 start=__ pc()时，将返回 20（如果将第一个位置标记为 0，则第二个位置为 20；否则如果从 1 开始标记，则第二个位置为 21），因为第一个例程占用 20 字节。下面使用图示来演示这一过程。

当 JVM 生成第一个例程时，pc()返回 0，如图 2.13 所示。

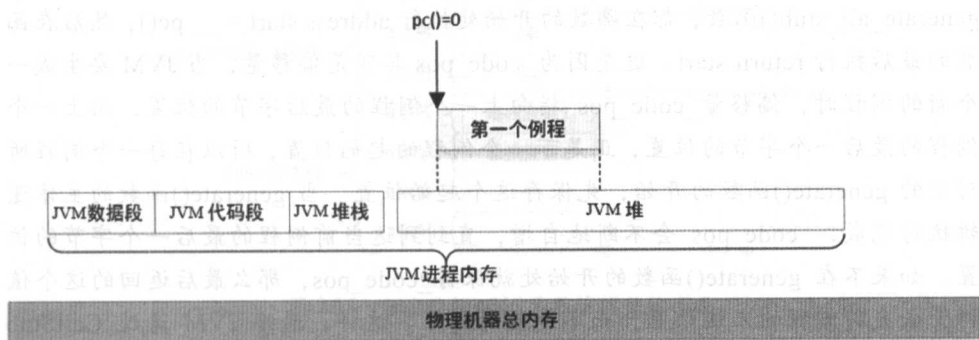


图 2.13 JVM 第一个例程的起始位置返回 0

（注：JVM 的例程都写入 JVM 的堆内存中，在 JVM 初始化时，会初始化一个容量足够大的堆内存，例程会写入堆中靠近起始的位置。当 Java 程序开始运行后，JVM 将 Java 类对象实例陆续写入堆内存中。在讲解 JVM 初始化的章节中会有关于堆内存的详细分析。）

JVM 中每一个例程都一个对应的 generate()函数（具体的函数名不同，但是基本都以 generate_ 开头），假设第一个例程占 20 个字节码，则当第一个例程所对应的 generate()函数执行完成后，_code_pos 会自动累加到 20，于是当 JVM 生成第二个例程时，pc()就会返回 20，如

图 2.14 所示。

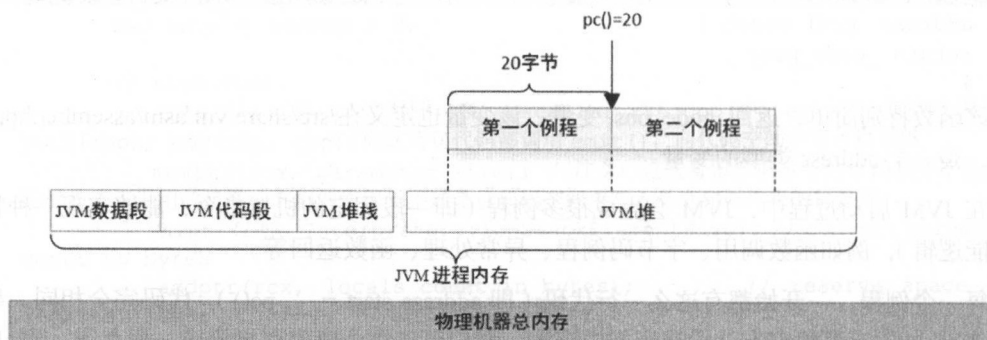


图 2.14 JVM 例程的起始位置

通过图 2.13 和图 2.14 可知, JVM 每生成一个例程, 就会将例程起始位置增加, 每一个例程都会占用 JVM 堆内存的一块连续区域, 相邻例程之间的内存区域相连 (即内存位置是靠在一起的), 所有的例程最后连成一块连续的区域。而事实上, JVM 内部的确是这样来划分内存的, 后面会详细讲。

注意: 在每一个例程所对应的 `generate()` 函数内部, 例如这里的 `generate_all_stub()` 函数, 都在函数的开始处执行 `address start = __pc()`, 然后在函数的最后执行 `return start`。这是因为 `_code_pos` 其实是偏移量, 当 JVM 要生成一个新的例程时, 偏移量 `_code_pos` 指向上一个例程的最后字节的位置, 而上一个例程的最后一个字节的位置, 正是下一个例程的起始位置, 所以在每一个例程所对应的 `generate()` 函数的开始, 先保存这个起始位置。当 `generate()` 函数的主体逻辑执行完成, `_code_pos` 会不断地自增, 直到到达当前例程的最后一个字节的位置。如果不在 `generate()` 函数的开始处就保存 `_code_pos`, 那么最后返回的这个值将变成当前例程的末端位置, 而不是起始位置。这样, 最终 JVM 通过 `CallStub` 这个函数指针 (或者其他例程所对应的函数指针) 来执行这段动态生成的机器指令, 就会因为位置不对而报错。

2. 定义入参

`generate_all_stub()` 接下来执行下面一段代码:

```
assert(frame::entry_frame_call_wrapper_offset == 2, "adjust this code");
bool sse_save = false;
const Address rsp_after_call(rbp, -4 * wordSize); // same as in
generate_catch_exception()!
```



```

const int    locals_count_in_bytes  (4*wordSize);
const Address mxcsr_save    (rbp, -4 * wordSize);
const Address saved_rbx    (rbp, -3 * wordSize);
const Address saved_rsi    (rbp, -2 * wordSize);
const Address saved_rdi    (rbp, -1 * wordSize);
//...

```

在讲解入参之前，我们先回顾下在前面分析机器调用时的调用者和被调用者堆栈模型。当时是以 `main()` 和 `add()` 举例说明的（可以回头再看看汇编代码和内存模型图），`main()` 是调用者函数 caller，`add()` 是被调用函数 callee，当物理机器由 caller 执行到 callee 时，堆栈模型的变化如图 2.15 所示。

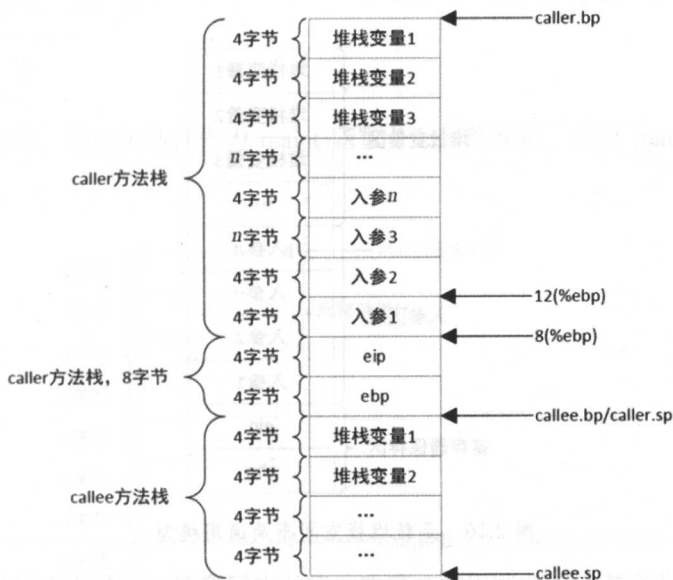


图 2.15 `main()` 与 `add()` 函数堆栈整体布局

由图 2.15 可知，一个函数的堆栈空间大体上可以分为 3 部分。

1) 堆栈变量区

保存方法的局部变量，或者对数据的地址引用（指针）。

如果一个方法中并没有局部变量，则编译器不会为该方法分配堆栈变量区。

2) 入参区域

如果当前方法调用了其他方法，并且给其他方法传递了参数，那么这些入参会保存在调用者的堆栈中，这就是所谓的“压栈”。

至少在 x86 平台上，入参区域相对于方法的堆栈变量区，在内存上位于低位置，即堆栈变量区在高地址方向，而入参区域则在低地址方向。x86 在分配堆栈空间时，本来就是按照由高地址向低地址的方向分配的。

3) ip 和 bp 区

ip 和 bp，一个是代码段寄存器，一个是堆栈栈基寄存器。这两个寄存器，一个用于恢复调用者方法的代码位置，一个用于恢复调用方法的堆栈位置，是完成物理机器函数调用机制的最主要的 2 个寄存器。关于这两个寄存器如何被保存，如何被恢复，在前面的章节中已经详细描述过。

函数堆栈空间的一般布局如图 2.16 所示。

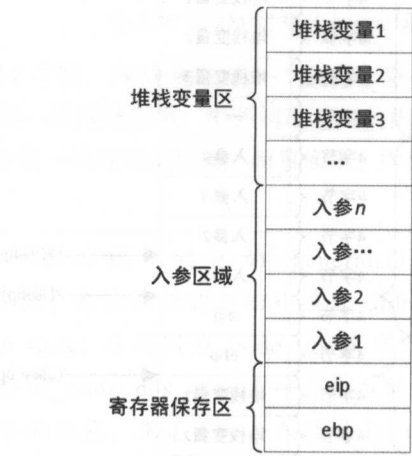


图 2.16 函数堆栈空间布局通用模型

其实，物理机器在执行函数调用时，存在一定的空间浪费。入参往往同时是当前方法的局部变量，编译器会将局部变量分配在局部变量区域，而在入参时，会将局部变量再次复制一份放到压栈区域，同一份数据被分配了两次堆栈空间，其实依靠编译器的智能性，完全可以将堆栈中的入参区域去掉。但是话又说回来，正因为编译器规定了入参空间分配的原则，并使入参按照从左到右或从右到左的顺序压栈，这种规范不仅仅让被调用函数在访问入参时享受到了极大的便利，也让 JVM 设计者（詹爷）在设计 Java 函数调用机制时，能够基于这一规范，随心所欲地发挥。很难想象，如果缺少了这一规范，jvm，包括很多其他虚拟机，在设计函数调用机制时，会是怎样的一种场景，大家会想出多少招式来实现这一复杂的逻辑？

基于这一规范，在被调用者方法中，访问入参，就变得有规律可循。下面先看一个简单的例子，假设有如下一个 C 程序：

清单：示例程序

作用：C 程序示例

```
int add(int, int);
int main(){
    int a = 6;
    int b = 8;

    // ...定义若干局部变量

    int c=add(a, b);
    return 0;
}

int add(int x, int y){
    int z=x+y;
    return z;
}
```

当该程序运行时，物理机器首先为 main()函数分配局部变量，此时 main()函数堆栈内存布局如图 2.17 所示。

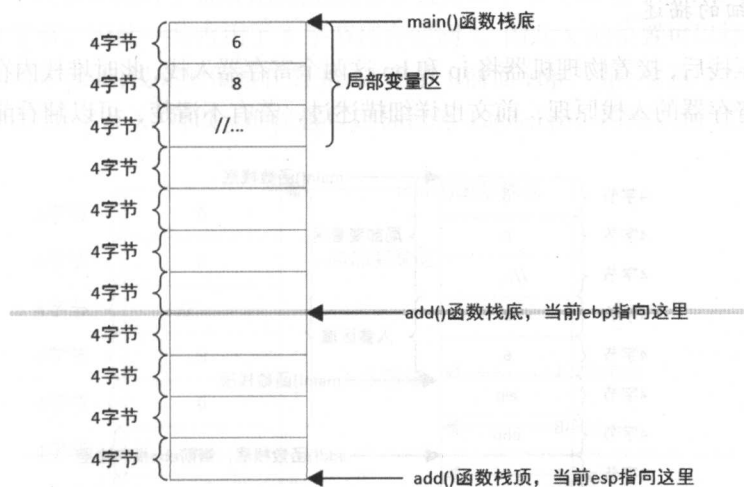


图 2.17 main()函数堆栈布局

(注：细心的读者会发现这个图有问题，因为当 CPU 运行 main()函数指令时，根本不会为 add()函数分配堆栈空间，因此这里将 add()函数的堆栈空间画上去是错误的。作者将其添加上去，主要是为了让读者能够总览调用者与被调用者之间的堆栈布局，从而建立这种印象。)

main()函数开始调用 add()函数之前，main()函数会将入参压栈，注意：参数被压到 main()函数的堆栈中。此时整体堆栈空间布局如图 2.18 所示。

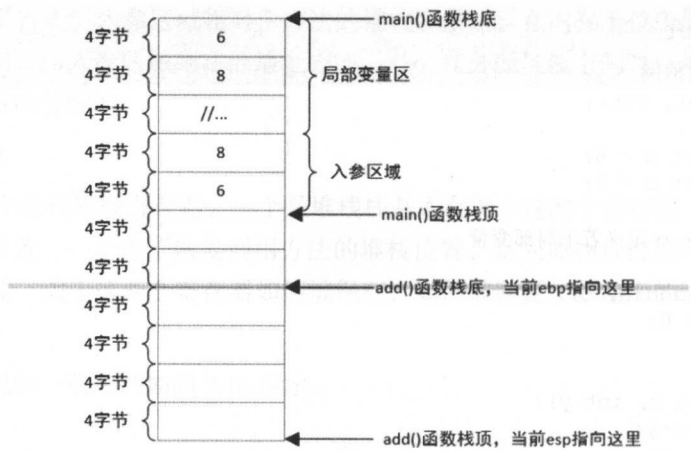


图 2.18 `main()`函数压参后的堆栈整体布局

注意：在 x86 平台上，入参以逆向顺序压栈，因此 8 先压栈，6 后压栈。关于参数压栈的机器指令，大家可以回头看前面讲解物理机器函数调用机制的章节，里面有详细的描述。

完成入参压栈后，接着物理机器将 `ip` 和 `bp` 这两个寄存器入栈，此时堆栈内存布局如图 2.19 所示（这两个寄存器的入栈原理，前文也详细描述过，若有不清楚，可以翻看前面章节）。

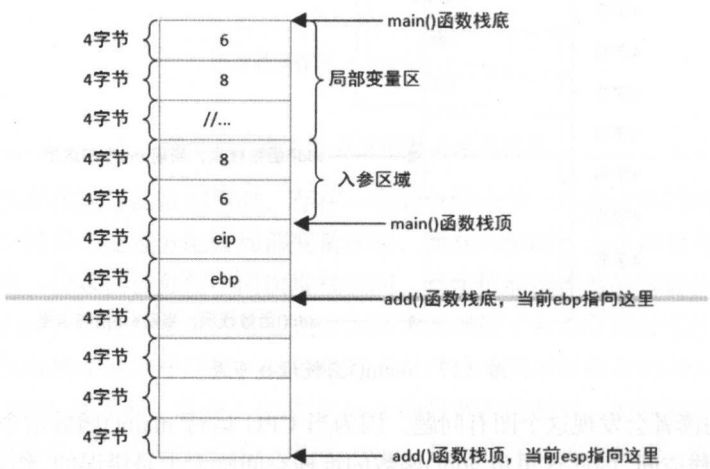


图 2.19 `ip` 和 `bp` 寄存器压栈

到了这一步，完成了 `ip` 和 `bp` 寄存器的入栈，物理机器就开始为被调用函数 `add()` 分配堆栈空间了，`add()` 函数的栈底与 `ebp` 寄存器相邻。在 x86 平台上，堆栈由高内存地址往低内存地址

方向分配，因此 `add()` 函数的栈底相对于 `ebp` 寄存器而言，处于低内存位置。

前面讲过，物理机器对堆栈内存寻址的方式为相对偏移，可以通过相对栈底 `bp` 或栈顶 `sp` 的位置来绝对定位堆栈内存位置。当相对 `sp`（栈顶）进行寻址时，如果对位于 `sp` 下方（低内存地址方向）的堆栈内存进行寻址，则使用 `-m(%sp)` 这样的方式，其含义是 `sp` 寄存器的值减去 `m` 字节，`m` 是个自然数，例如 `-3(%sp)`。反之，如果对位于 `sp` 上方（高内存地址方向）的堆栈内存进行寻址，则使用 `m(%sp)` 这样的方式，其含义是 `sp` 寄存器的值加上 `m` 字节，例如 `3(%sp)`。

相对于 `bp`（栈底）进行相对寻址，也是同样的道理。

接着上面的例子，当物理机器执行到 `add()` 函数时，`add()` 函数要执行 `int z = x + y` 这样的代码，就必须读取入参。此时物理机器的 `bp` 指向 `add()` 函数的栈底，因此可以通过 `add()` 函数堆栈栈底进行相对寻址，读取两个入参 `x` 和 `y`。那么 `x` 和 `y` 相对 `add()` 函数栈底的偏移量是多少呢？

我们来进行看图说话，看图 2.19，`ebp` 相对于 `add()` 栈底的偏移位置是 0，因此可以标记为 `(%ebp)`。`eip` 相对于 `add()` 栈底的偏移位置是 4 字节（32 位平台，`ebp` 寄存器占了 4 字节的空间），同时，`eip` 相对于 `add()` 函数栈底，位于高内存地址方向，因此标记为 `4(%ebp)`。

同样，第一个入参 `x` 相对于 `add()` 栈底的偏移位置是 8 字节（32 位平台，`ebp` 和 `eip` 这 2 个寄存器各占去 4 字节，因此一共占用了 8 字节内存空间），因此 `x` 的位置可以标记为 `8(%ebp)`。第二个入参 `y` 相对于 `add()` 栈底的偏移位置是 12 字节（前面 `ebp`、`eip`、`x` 这 3 个数据各占 4 字节，因此一共占去 12 字节），所以 `y` 的位置可以标记为 `-12(%ebp)`。用图 2.20 显示这种标记。

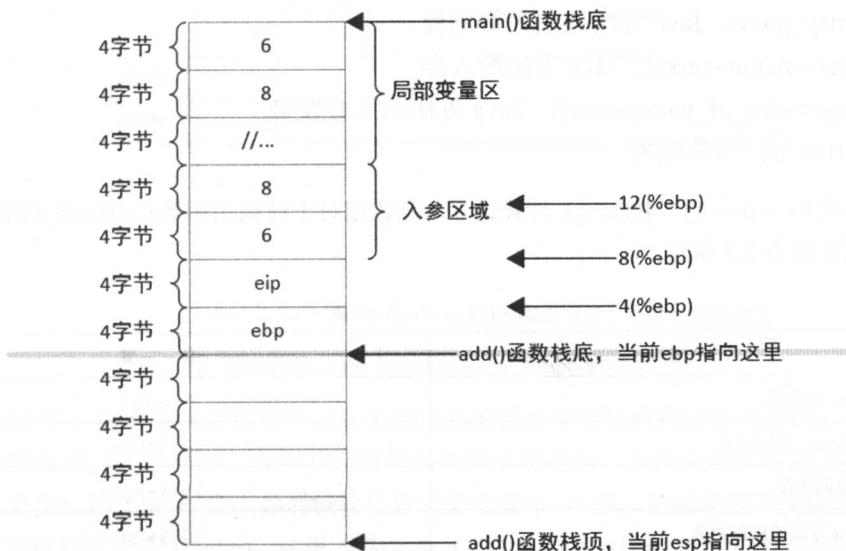


图 2.20 以 `bp` 为基址对变量进行标记

现在，假设 `main()` 函数在调用 `add()` 时，不仅仅包含 2 个人参，而是包含 3 个，那么我们根据图 2.20，可以推导出这 3 个人参的内存定位：

- ◎ 第 1 个人参为 `8(%ebp)`。
- ◎ 第 2 个人参为 `12(%ebp)`。
- ◎ 第 3 个人参为 `16(%ebp)`。

根据这种推导关系，使用数学归纳法，可以得到这样的人参寻址公式：

$$P_n = (n + 1) * 4(\%ebp)$$

在该公式中， n 表示第 n 个人参（按从左至右的顺序）， P_n 表示第 n 个人参的位置。

有了这个公式，我们在理解 JVM 调用 `CallStub` 函数指针时，就会理解 JVM 的人参定义。那么现在就让我们将目光再次定格到 `generate_call_stub()` 这个函数中。由于 `CallStub` 函数指针最终指向 `generate_call_stub()` 这个函数所返回的一段机器指令，因此 `generate_call_stub()` 中生成的机器指令其实就是被调用函数对应的机器码，可以将其理解成 `CallStub()` 的函数体。

JVM 在 `javaCalls.cpp::call_helper()` 函数中执行 `CallStub` 调用，在调用时，传递了如下 8 个参数：

- ◎ `(address)&link`，连接器
- ◎ `result_val_address`，返回地址
- ◎ `result_type`，返回类型
- ◎ `method()`，Java 方法的内部对象
- ◎ `entry_point`，Java 方法调用入口例程
- ◎ `args->parameters()`，Java 方法的人参
- ◎ `args->size_of_parameters()`，Java 方法的人参数量
- ◎ `CHECK`：当前线程

按照公式 $P_n = (n + 1) * 4(\%ebp)$ ，计算这 8 个参数相对于被调用函数 `CallStub()` 栈底的位置，计算后的位置如表 2.2 所示。

表 2.2 入参位置

入 参	位 置
<code>(address)&link</code> ：连接器	<code>8(%ebp)</code>
<code>result_val_address</code> ：返回地址	<code>12(%ebp)</code>
<code>result_type</code> ：返回类型	<code>16(%ebp)</code>
<code>method()</code> ：Java 方法的内部对象	<code>20(%ebp)</code>
<code>entry_point</code> ：Java 方法调用入口例程	<code>24(%ebp)</code>

续表

入 参	位 置
args->parameters(): Java 方法的入参	28(%ebp)
args->size_of_parameters(): Java 方法的入参数量	32(%ebp)
CHECK: 当前线程	36(%ebp)

当 JVM 进入到 CallStub 这个函数指针所代表的函数的堆栈中后，调用者 javaCalls::call_helper 与被调用者 CallStub()之间的堆栈内存布局如图 2.21 所示。

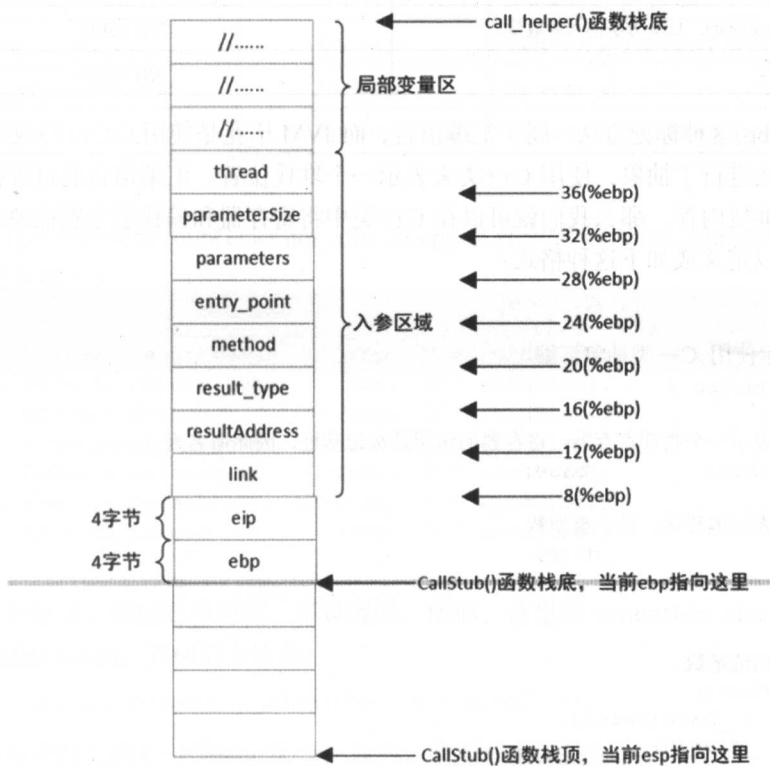


图 2.21 javaCalls::call_helper()与 CallStub()的堆栈布局

在图 2.21 中，这 8 个入参相对于 ebp 的偏移量都是以字节为单位进行计算的，在 32 位平台上一个指针占 32 位，4 字节，如果我们将偏移量以 4 字节为一个单位进行标记，而不是以 1 字节为计量单位，那么这 8 个入参的偏移位置的值会缩小 4 倍，看起来就不会那么大，例如 28(%ebp)就会被标记为 7N(%ebp)，这里 N 为常量 4。以 4 字节为单位对这 8 个入参的偏移位置进行重新标记，如表 2.3 所示。

表 2.3 对 CallStub() 的 8 个入参的 ebp 偏移位置按 4 字节为单位进行标记

入 参	位 置
(address)&link: 连接器	2N(%ebp)
result_val_address: 返回地址	3N(%ebp)
result_type: 返回类型	4N(%ebp)
method(): Java 方法的内部对象	5N(%ebp)
entry_point: Java 方法调用入口例程	6N(%ebp)
args->parameters(): Java 方法的入参	7N(%ebp)
args->size_of_parameters(): Java 方法的入参数量	8N(%ebp)
CHECK: 当前线程	9N(%ebp)

诸如 3(%ebp) 这种标记方法，属于汇编语言，而 JVM 毕竟是使用 C/C++ 写成的，于是 JVM 大神对汇编语法进行了抽象，使用 C++ 类来表示一个堆栈位置。汇编语言通过寄存器和偏移量唯一定位一个堆栈内存，那么我们就可以在 C++ 类中将寄存器和偏移量分别抽象成两个变量，这个 C++ 类可以定义成如下这种格式：

清单：test

作用：演示使用 C++ 类抽象汇编

```
class Address {
private:
    // 表示一个物理寄存器，寄存器的作用是标记基址，因此命名为_base
    Register    _base;

    // 表示偏移量，是个整型数
    int         _disp;

public:
    // 构造函数
    Address()
        : _base(noreg),
          _disp(0)
    {
    }
}
```

该程序里有一个 Register 类，先不用关心其具体的数据结构，只需知道它能够表示一个物理寄存器就够了。使用 C++ 类对汇编堆栈寻址进行抽象后，便可以直接用该类进行堆栈寻址了。假设要对 8(%ebp) 进行寻址，可以这样写：

```
Address position (rbp, 8);
```

这就声明了一个 C++ 类对象 position，最终可以通过该对象，还原出汇编指令 8(%ebp)。

基于该 C++ 类, 我们可以这样对 CallStub() 的 8 个入参的偏移位置进行标记(如表 2.4 所示)。

表 2.4 对 CallStub() 的 8 个入参的 ebp 偏移位置按 4 字节为单位进行标记

入 参	位 置	C++类标记
(address)&link	2N(%ebp)	Address link(rbp, 2N)
result_val_address	3N(%ebp)	Address returnAddress(rbp, 3N)
result_type	4N(%ebp)	Address resultType(rbp, 4N)
method()	5N(%ebp)	Address method(rbp, 5N)
entry_point	6N(%ebp)	Address entryPoint(rbp, 6N)
args->parameters()	7N(%ebp)	Address parameters(rbp, 7N)
args->size_of_parameters()	8N(%ebp)	Address parametersSize(rbp, 8N)
CHECK	9N(%ebp)	Address thread(rbp, 9N)

(注: 表中的 $N=4$)

现在, 让我们再回到 stubGenerator_x86_32.cpp 文件中的 generate_call_stub() 函数开始的那 10 几行的类声明 ():

```
//...
const Address result      (rbp, 3 * wordSize);
const Address result_type (rbp, 4 * wordSize);
const Address method      (rbp, 5 * wordSize);
const Address entry_point (rbp, 6 * wordSize);
const Address parameters   (rbp, 7 * wordSize);
const Address parameter_size (rbp, 8 * wordSize);
const Address thread       (rbp, 9 * wordSize);
//...
```

对于这个定义, 相信聪明的你, 不难理解。例如, 这里的 parameters_size 代表的堆栈位置是 $8 * wordSize(\%rbp)$, 其标记方法是:

```
const Address parameter_size(rbp, 8 * wordSize);
```

这正好与我们上面 C++ 类标记法是一致的, 我们在上文标记为:

```
Address position (rbp, 8N);
```

事实上, JVM 里面所定义的 Address 类与我们在上文自定义的 Address 类并无本质区别, 无非是额外多了几个变量而已。同时, JVM 里定义了常量 wordSize, 其声明如下:

清单: /src/share/vm/utilities/globalDefinitions.hpp

作用: wordSize 常量定义

```
const int wordSize = sizeof(char*);
```

在 32 位平台上, `sizeof(char*)` 将返回 4; 在 64 位平台上, `sizeof(char*)` 返回 8。由于 `char*` 是一个指针类型, 而指针能够指向物理机器内存的任何一个地址, 因此, 在 N 位平台上, 指针的宽度必须也至少是 N 位, 这样指针才能寻址到内存任何一个位置。假如内存总大小是 64 比特, 那么指针宽度只需 6 位, 即可寻址到内存任何位置, 2 的 6 次方正好等于 64。JVM 作为一款能够兼容大部分主流操作系统的虚拟机, 兼容指针长度是其基本功。

上面这段话, 对新手起到一个启发思路的作用。我们还是回到 `generate_call_stub()` 函数一开始的变量定义, 要注意, 你看不到一个类似于

`const Address linke (rbp, 2 * wordSize)` 这样的定义, 这是因为这个函数中并没有用到这个入参。

同时, 你还会看到如下定义:

```
//...
const Address mxcsr_save    (rbp, -4 * wordSize);
const Address saved_rbx    (rbp, -3 * wordSize);
const Address saved_rsi    (rbp, -2 * wordSize);
const Address saved_rdi    (rbp, -1 * wordSize);

const Address result        (rbp, 3 * wordSize);
//...
```

`mxcsr_save`、`saved_rbx`、`saved_rsi`、`saved_rdi` 这 4 个位置相对于 `rbp` 的偏移量是负数, 这很容易理解, 说明这 4 个参数的位置在 `CallStub()` 函数的堆栈内部, 而不是位于调用函数 `javaCalls::call_helper()` 堆栈内, 所以相对于 `rbp` 的偏移量才会是负数。这 4 个变量用于保存调用者的信息, 在后面会详细分析这 4 个变量。

讲完了 `generate_call_stub()` 函数开始的那 10 几行的类声明, 接下来开始真正进入 `CallStub` 例程的逻辑分析。

3. CallStub: 保存调用者堆栈

`generate_call_stub()` 函数的逻辑部分从下面这行代码开始:

```
__ enter();
```

这行代码在不同的硬件平台上, 对应不同的机器指令。在 x86 平台上, 其函数定义在 `assembly_x86.cpp` 文件中, 定义如下:

清单: `/src/cpu/x86/vm/assembly_x86.cpp:enter()`

作用: `enter()` 函数定义

```
void MacroAssembler::enter() {
    push(rbp);
    mov(rbp, rsp);
}
```

这两条指令最终会在 JVM 运行期被翻译为如下所示的对应的机器指令：

```
push %bp
mov %sp, %bp
```

如果你认真地看过前面章节，或者熟悉汇编指令，那么你对这两条指令一定不陌生。在 x86 平台上，物理机器调用任何一个函数之前，都会执行这两条指令，`push %ebp` 指令的含义是保存调用者函数的栈基地址，`mov %sp, %bp` 指令的含义是重新指定栈基地址。由于即将开始新的函数，因此需要将栈基指向调用者函数的栈顶位置，调用者函数的栈顶位置就是被调用者函数的栈基位置（严格来说这句话是错误的，因为调用者与被调用者函数之间还隔着两个寄存器：`ip` 和 `bp`）。

执行 `enter()` 之前，`bp` 和 `sp` 这 2 个寄存器指向位置如图 2.22 左半部分所示，此时堆栈空间属于调用者函数 `javaCalls::call_helper()`。执行 `enter()` 之后，这 2 个寄存器将指向新的函数，如图 2.22 右半部分所示，此时堆栈空间属于被调用者函数 `CallStub()`：

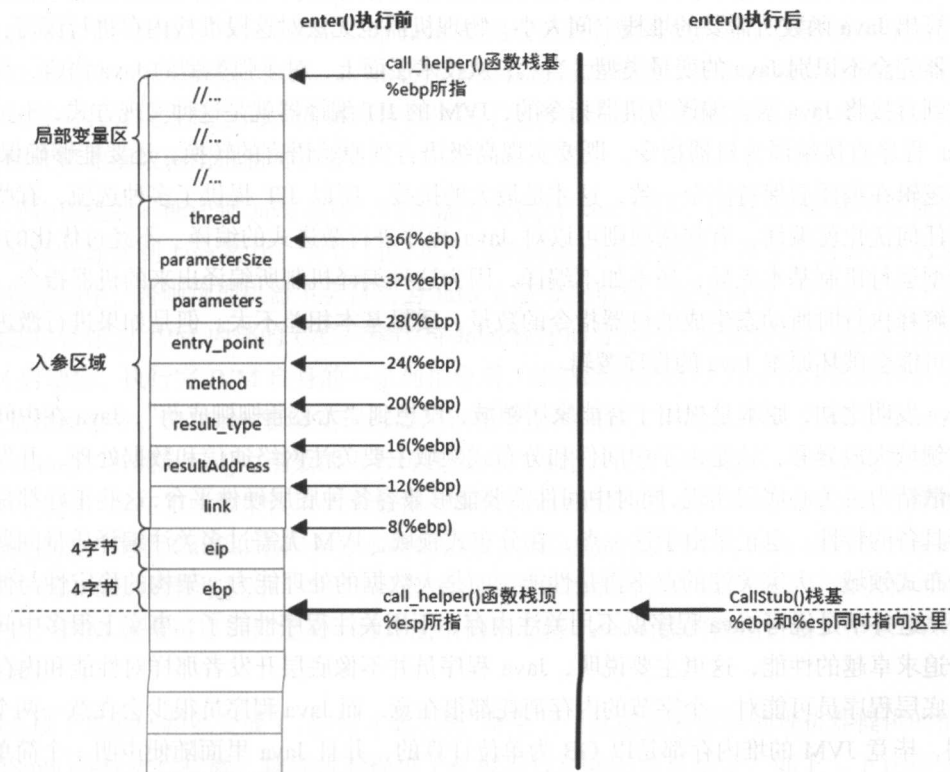


图 2.22 `javaCalls::call_helper()` 执行 `enter()` 前后 `sp` 和 `bp` 寄存器指向

4. CallStub: 动态分配堆栈

应该说, JVM 之所以能够在物理机器上分配 Java 语言变量类型的堆栈, 完全得益于机器级别对堆栈空间分配的指令支持。JVM 充分利用了这一点, 通过重新分配堆栈空间, 从而为 JVM 调用 Java 函数奠定基石。

大家正常写一段 C 程序, 编译时, 编译器会根据被调用函数中的变量声明, 自动计算出被调用函数需要多大的堆栈空间。最终物理机器在执行这个 C 程序时, 将按照编译器所计算出的大小为被调用函数分配堆栈空间。

而 Java 程序由于无法直接被编译为机器指令, 因此 Java 编译器无法直接计算一个 Java 函数需要多大的堆栈空间 (其实如果真的要说的话, 也是可以做到的), 但是如果仅仅解决 Java 函数的堆栈空间的自动计算, 还是无法实现 Java 函数的调用, 因为 Java 有自己的变量类型, 这些变量类型不像 C 语言的变量, Java 的变量类型并不能直接被编译成物理机器所识别的数据类型, 而 C 语言变量最终完全被编译成物理机器所识别的类型。所以, 即使 Java 程序编译器能够自动计算出 Java 函数所需要的堆栈空间大小, 物理机器也无法对这段堆栈内存进行读写, 因为物理机器完全不识别 Java 的变量类型。当然, 从技术层面上, 对于强类型的 Java 语言, 是完全可以做到直接将 Java 语言编译为机器指令的, JVM 的 JIT 编译器就是这种实现方式, 不过由于将 Java 程序直接编译为机器指令, 既要实现高级语言到原始语言的转换, 还要能够确保 Java 程序的逻辑在编译后保持完全一致, 这才是最大的挑战。所以 JIT 提供了多种选项, 有些选项不进行任何优化就编译, 有些选项则可以对 Java 程序进行激进式的编译。不经过优化的编译, 与解释型运行机制基本无异, 还不如不编译, 因为这种编译机制所编译出来的机器指令, 与直接进行解释执行时所动态生成的机器指令的数量、质量基本相差不大。但是如果进行激进式的编译, 可能会破坏原本 Java 的程序逻辑。

Java 发明之初, 原本是想用于智能家居领域, 没想到“无心插柳柳成荫”, Java 在中间件和分布式领域大放异彩, 这是由于中间件和分布式领域主要关注网络通信和数据处理, 开发者不用再分散精力去关心底层实现, 同时中间件需要能够兼容各种底层硬件平台, 这些正好都是 Java 天生所具备的特性。也正是由于这一点, 在分布式领域, JVM 无需过多关注编译质量问题, 毕竟在分布式领域, 大家关注的点不再是性能, 而是大数据的处理能力、架构的稳定性与伸缩性等方面 (这并不是说写 Java 程序就不用关注内存、不用关注程序性能了, 事实上很多中间件反而十分追求卓越的性能, 这里主要说明, Java 程序员并不像底层开发者那样对性能和内存十分关注, 底层程序员可能对一个字节的内存消耗都很在意, 而 Java 程序员很少会在意一两个字节的占用, 毕竟 JVM 的堆内存都是以 GB 为单位计算的, 并且 Java 里面随便声明一个简单类型的变量, 都会占用超出一个字节的内存)。但是在移动操作系统领域, 个人更倾向于一种能够直接将 Java 程序 (或类似 Java 的编程语言所写出来的程序) 编译成本地物理机器指令的机制, 或

者一种能够直接运行 JVM 字节码指令的 CPU（有公司在做这方面的研究），从而完全消除虚拟机依赖，这样既能确保商业项目对效率的极致追求，又能保证 Java 程序在本地运行时对性能的苛刻需求。

上面仅是一家之言，相信大家定有不同观点。上面扯得有点远了，回归主题。JVM 为了能够调用 Java 函数，需要在运行期知道一个 Java 函数的入参大小，然后动态计算出所需要的堆栈空间。这就是 JVM 能够调用 Java 函数的核心机制。这里的关键问题是，CallStub()作为被 javaCalls::call_helper()调用的函数，JVM 通过 javaCalls::call_helper()最终调用到 Java 函数，JVM 作为一款使用 C/C++ 编写而成的程序，被编译后，C/C++ 编译器自然会计算出 javaCalls::call_helper()传递给 CallStub()的入参的空间大小，但是 C/C++ 编译器并不会因此就自动计算出 Java 函数的入参数量及所需内存空间大小，因为 C/C++ 编译器并不识别 Java 程序，并且在 JVM 被编译期间，JVM 尚未加载任何 Java 程序，因此 JVM 对 Java 程序完全无感。等到 JVM 程序运行起来后，JVM 会加载 Java 程序，并通过 javaCalls::call_helper()调用 Java 主函数。JVM 在执行 Java 函数调用时，仍然沿用了物理机器所使用的“堆栈”这一算法数据结构，并没有发明新的轮子，因此 JVM 仍然需要为 Java 被调用函数分配堆栈内存，保存被调用函数的局部变量以及相关上下文数据。因此，JVM 必然需要在运行期动态计算 Java 被调用函数的空间大小，并动态为其分配堆栈空间。而物理机器提供了这种动态分配堆栈空间的能力，这一点在前文讲述物理机器函数调用机制时讲过。

由于物理机器不能识别 Java 程序，也不能直接执行 Java 程序，因此 JVM 必然要通过自己作为一座桥梁连接到 Java 程序，并让 Java 被调用的函数的堆栈能够“寄生”在 JVM 的某个函数的堆栈空间中，否则物理机器不会自动为 Java 方法分配堆栈。前面讲过，JVM 选择 CallStub 这一函数指针作为 JVM 内部的 C/C++ 程序与 Java 程序的分水岭，或者桥梁，通过这座桥梁，当 JVM 启动后，执行完 JVM 自身的一系列指令后，能够跳转到执行 Java 程序经翻译后所对应的二进制机器指令，CallStub 能够实现机器逻辑指令上的联接，同时，JVM 会调用 Java 的入口主函数 main()，并将 main()主函数的入参传递进去。因此，在分水岭之后，JVM 需要为主函数分配堆栈空间，以在主函数中读取入参数据。那么 Java 函数所需要的堆栈空间分配在哪里呢？答案是明显的，既然 CallStub()作为分水岭的函数，很自然地，JVM 将 Java 函数堆栈空间“寄生”在了 CallStub()函数堆栈中。当然，从技术实现的手段而言，JVM 并非一定要选择“寄生”这种方式，JVM 完全可以另外定义一种算法结构来支持 Java 函数的调用机制，但是 JVM 并没有这么做。

那么接下来的问题就变成了，如何才能实现“寄生”？这就需要依靠物理机器提供的指令，对 CallStub()堆栈进行扩展。物理机器提供了扩展堆栈空间的简单指令，如下（下述指令基于 x86 平台）：

```
sub operand, %sp
```

operand 是一个自然数，例如 8、16 或者其他数值。这条指令表示将堆栈向下扩展一定的空间。

如果你写了一个 C/C++ 程序，C/C++ 编译器会自动计算一个函数所需要的堆栈大小，例如下面这个例子：

清单：示例

作用：C 函数调用示例

```
#include<stdio.h>

int add(int x, int y);
int main(){
    int a=5;
    int b=3;

    int c=add(a,b);
    printf("%d\n",c);

    return 0;
}

int add(int x, int y){
    int z=x+y;
    return z;
}
```

将这段程序编译为汇编程序，得到：

清单：示例

作用：C 函数调用示例

```
main:
    pushl    %ebp
    movl     %esp, %ebp

    andl     $-16, %esp
    subl     $32, %esp

    movl     $5, 20(%esp)
    movl     $3, 24(%esp)

    movl     24(%esp), %eax
    movl     %eax, 4(%esp)
    movl     20(%esp), %eax
    movl     %eax, (%esp)

    call     add
```

```
movl%eax, 28(%esp)
```

```
movl28(%esp), %edx
```

```
movl%edx, 4(%esp)
```

```
movl%eax, (%esp)
```

```
callprintf
```

```
movl$0, %eax
```

```
leave
```

```
ret
```

```
add:
```

```
pushl %ebp
```

```
movl%esp, %ebp
```

```
subl$16, %esp
```

```
movl12(%ebp), %eax
```

```
movl8(%ebp), %edx
```

```
addl%edx, %eax
```

```
movl%eax, -4(%ebp)
```

```
movl-4(%ebp), %eax
```

```
leave
```

```
ret
```

对于被调用函数 add(), 其内部只声明了一个局部变量 z。对于入参 x 和 y, 在所对应的汇编程序中, 并没有发现编译器为其分配堆栈空间, 编译器将使用 ax 和 dx 这两个寄存器分别保存这 2 个入参。因此, add() 函数其实只需要为变量 z 分配堆栈空间, 大小为 32 字节。由于编译器会自动对齐, 因此最终编译器为 add() 函数分配了 16 字节的堆栈, 分配的方式如下:

```
subl $16, %esp
```

执行这段汇编程序后, 最终系统打印出正确的结果值: 8。

虽然 C/C++ 编译器会自动计算堆栈大小, 但是可以人工对计算的结果值进行修改。我们继续实验, 将上面这段汇编程序中 add 代码段的堆栈空间变为 64, 修改后的程序如下:

清单: 示例

作用: C 函数调用示例

```
main:
```

```
pushl %ebp
```

```
movl%esp, %ebp
```

```
andl$-16, %esp
```

```
subl$32, %esp
```



```
    movl$5, 20(%esp)
    movl$3, 24(%esp)

    movl24(%esp), %eax
    movl%eax, 4(%esp)
    movl20(%esp), %eax
    movl%eax, (%esp)

    calladd
    movl%eax, 28(%esp)

    movl28(%esp), %edx
    movl%edx, 4(%esp)
    movl%eax, (%esp)
    callprintf
    movl$0, %eax

    leave
    ret

add:
    pushl    %ebp
    movl%esp, %ebp
    subl$64, %esp

    movl12(%ebp), %eax
    movl8(%ebp), %edx
    addl%edx, %eax
    movl%eax, -4(%ebp)
    movl-4(%ebp), %eax

    leave
    ret
```

注意，现在 add 标段中 `subl $16, %esp` 变成了 `subl $64, %esp`，这表示为 add 分配 64 字节空间。运行修改后的汇编程序，会发现程序依然输出了正确的结果：8。

这就是 JVM 实现堆栈“寄生”的机制。扩展别人的堆栈，存储自己所需要的数据。

CallStub() 作为 JVM 内部 C/C++ 与 Java 程序的分水岭，CallStub() 调用者 `javaCalls::call_helper()` 并没有直接将 Java 函数的入参传递给 CallStub()，因为这个调用者并不直接是 Java 函数自己，因此在 JVM 的编译阶段，并没有将 Java 函数压栈。在上面 C 示例程序中，`main()` 函数调用了 `add()` 函数，`add()` 函数的 2 个入参 `x` 和 `y`，在 `main()` 函数中完成了压栈，这 2

个人参实际被保存在了 main() 这个调用函数堆栈中。同理，JVM 要调用 Java 主函数（由于 JVM 第一次调用 Java 函数从 Java 程序的主函数 main() 开始，因此这里以 main() 为例讲述）main()，众所周知，Java 主函数 main() 包含一个字符串数组入参，因此 Java 主函数的声明格式一定如下：

清单：示例

作用：Java 主函数声明示例

```
public static void main(String[] args){
    //...
}
```

Java 主函数一定包含一个 String[] 类型的入参。既然 JVM 会调用这个方法，那么按照 C/C++ 程序的函数调用机制，JVM 中调用 Java 程序主函数 main() 的函数（为了表述方便，我们假设 JVM 中调用 Java 主函数 main() 的函数名为 xxx()），必然要对 Java 主函数 main() 的入参 String[] args 进行压栈，JVM 会将 args 参数保存在 xxx() 的堆栈中，这样在 Java 的主函数 main() 内部才能访问入参数据。

但是，由于 JVM 在编译期间对 Java 程序完全“无感”，JVM 在编译时，压根儿就不知道加载的是什么样的 Java 程序，也不知道 Java 的主函数的入参数据是什么，因此 C/C++ 编译器在编译 JVM 时，对于调用 Java 主函数 main() 的 xxx() 函数，并不会将 Java 主函数 main() 所需的入参 String[] args 数据压栈到 xxx() 函数中，xxx() 函数中根本就没有 args 数据。

那么问题来了，当 JVM 执行完自己的一系列指令后，最终开始调用 Java 的主函数 main() 时，Java 主函数 main() 所需的 args 入参信息保存在哪里，从哪里获取，这些信息又是什么呢？

一切奥秘都在 CallStub 这个函数指针中所指向的函数中，即上文一直在讲述的 stubGenerator_x86_32 : generate_call_stub() 函数。为了讲述方便，前文一直直接使用 CallStub() 来指代 generate_call_stub() 函数，实际上 JVM 内部并不存在 CallStub() 这个函数，CallStub 仅仅是一个函数指针。但是为了讲述方便，下文继续使用 CallStub() 这一假想中的函数。

前面说了很多次，CallStub() 是 JVM 内部 C/C++ 程序与 Java 程序之间的分水岭和桥梁，分水岭的其中一个重要作用就是能够将 Java 程序被调用的函数的入参分配到堆栈中，这样在 Java 函数中才能对 Java 类型的入参进行寻址。其实，刚才所假想的在 JVM 内部调用 Java 程序主函数 main() 的 xxx() 函数，就是 CallStub() 函数。

刚才讲到，既然 CallStub() 函数调用了 Java 程序的主函数 main()，那么在 CallStub() 函数中必然要将 Java 程序主函数 main() 所需的入参信息 String[] args 进行压栈，否则 Java 主函数 main() 内部无法对入参进行寻址。但是在 JVM 编译期间，C/C++ 根本就不知道 Java 程序的任何信息，更无从谈起将 Java 主函数入参压栈。这个问题如何解决呢？那就是使用动态分配堆栈的方式，或者“寄生”。CallStub() 函数所对应的机器指令是在 JVM 启动过程中动态生成的，而非编译期

间生成，在 `CallStub()` 内部需要知道被调用的 Java 函数的入参数量，并依此计算入参所需空间大小，最终将其压栈，这样当 JVM 在执行 Java 函数时，在被调用的 Java 函数中就能对入参进行寻址。

在 `CallStub()` 函数（即 `stubGenerator_x86_32 : generate_call_stub()`）中，通过如下指令计算出被调用 Java 函数的入参数量，并保存调用者数据段现场：

清单：/src/cpu/x86/vm/stubGenerator_x86_32.cpp

作用：generate_call_stub() 函数

```
address generate_call_stub(address& return_address) {
    // 定义堆栈位置变量
    //...
    bool sse_save = false;
    const Address rsp_after_call(rbp, -4 * wordSize); // same as in generate_catch_exception()!
    const int locals_count_in_bytes (4*wordSize);
    const Address mxcsr_save (rbp, -4 * wordSize);
    //...

    // stub code
    __enter();
    __movptr(rcx, parameter_size);
    __shlptr(rcx, Interpreter::logStackElementSize);
    __addptr(rcx, locals_count_in_bytes);
    __subptr(rsp, rcx);
    __andptr(rsp, ~(StackAlignmentInBytes));

    //...
}
```

这段代码中的加粗部分，就是 JVM 在对被调用的 Java 函数的入参进行计算。`parameter_size` 是在 `generate_call_stub()` 函数开始处定义的堆栈变量，其定义如下：

```
const Address parameter_size(rbp, 8 * wordSize);
```

这个变量指向 `bp` 栈基往高地址偏移 8 个字长的位置，一个字长占 4 字节（32 位平台），因此实际相对 `bp` 的偏移量为 32 字节。此时 `bp` 指向 `CallStub()` 函数的栈底，`parameter_size` 指向位置如图 2.23 所示。

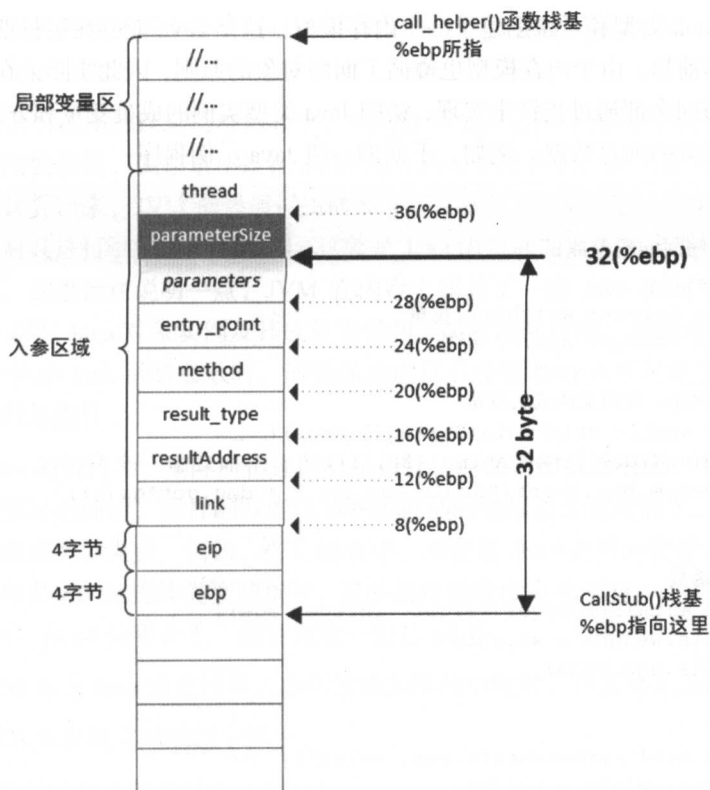


图 2.23 parameter_size 变量实际所代表的堆栈位置

javaCalls::call_helper()在调用 Java 函数之前，会读取 Java 函数的入参大小，Java 函数的入参大小由 Java 编译器在编译期间计算出来，因此 JVM 在执行 Java 函数之前，可以将其直接读取出来。JVM 得到 Java 函数所需要的入参数量后，便可以计算出入参压栈所需要的堆栈空间。

也许有人会疑惑，不同的入参，数据类型不尽相同，其所占内存大小肯定也不同，那么仅仅根据入参数量，如何就能够确定全部入参所需要的内存空间呢？例如，在 C 程序中，int 类型的入参和 char 类型的入参，其所占的内存大小一定不同，如果要计算全部入参空间大小，必须知道每一种数据类型所占的内存大小，然后进行累加求和。更何况 Java 语言的数据类型比 C 语言更加丰富，不同数据类型所占用的空间大小更加不同。

其实，众所周知，Java 是一门面向对象的编程语言，这一点不仅表现在 Java 的语义上，同时 JVM 在内存模型上也体现了这一原则。在 Java 语义上，定义任何类型的变量（除了 Java 的基本数据类型），都需要进行实例化，从而从语法层面上实现面向对象的宗旨。而 JVM 在内存

中，也为每一个 Java 类型和对象创建了一个内存模型，这是 Java 能够在运行期获取 Java 类型的描述信息的根本前提。由于内存模型也遵循了面向对象的原则，因此实际上在 JVM 内部，对 Java 类型实例的访问全部通过指针来实现，访问 Java 类型实例的成员变量和方法时，亦基于指针偏移量获取到对应的内存数据。例如，下面的一段 Java 示例程序：

清单：示例

作用：Java 程序面向对象测试

```
class Animal{
    private Integer weight;//重量
    private Integer age;// 年龄

    // 测试程序：访问类的成员变量
    public static void main(String[] args){
        Animal dog = new Animal(30, 1);
        System.out.print("dog's age is: " + dog.getAge());
    }

    // 构造函数
    public Animal(Integer weight, Integer age){
        this.weight = weight;
        this.age =age;
    }

    public void setWeight(Integer weight){
        this.weight = weight;
    }
    public Integer getWeight(){
        return this.weight;
    }

    public void setAge(Integer age){
        this.age = age;
    }
    public Integer getAge(){
        return this.age;
    }
}
```

在 JVM 执行 `Animal dog = new Animal(30, 1)` 时，会在 JVM 堆中为 dog 实例对象分配一段连续的内存空间，并根据构造函数所传入的数据对这段内存空间进行初始化，注意，这里是连续的内存空间。在 JVM 执行 `System.out.print("dog's age is: " + dog.getAge())` 访问 `dog.age` 成员变量时，实际上 JVM 将 Java 程序中的 dog 处理成了一个指针，通过该指针，JVM 可以找到在堆中所分配的 dog 实例数据。Animal 类型包含 weight 和 age 这 2 个类成员变量，并且类型都是 Integer，

因此 dog 在堆中的内存区域中持有对这两个对象的指针的引用，最终 JVM 通过 Integer 的指针获取到最终的 age 的值。

在 dog 实例所占用的连续的堆内存中，weight 和 age 这两个实例对象的指针相对于 dog 指针，具有不同的偏移量，偏移量不是在 JVM 运行期动态计算的，而是在 Java 程序的编译期，由编译器自动计算出来，JVM 最终通过指向 dog 实例的指针+偏移量，对 Java 类型的成员变量进行寻址，并对其进行赋值或取值操作。这就是 Java 语言面向对象的实现机制。后文会专门讲解具体的原理，这里旨在说明一点，JVM 在内存上建立了一套 Java 面向对象的标准模型，在 JVM 内部，一切对 Java 对象实例及其成员变量和成员方法的访问，最终皆通过指针得以寻址。同理，JVM 在传递 Java 函数参数时，所传递的也只不过是 Java 入参对象实例的指针而已。简而言之，传递的是指针。

正因为 Java 函数传参，实际所传递的只是指针，而在物理机器层面，不管何种数据类型的指针，其宽度都是相同的，指针的宽度仅与物理机器的数据总线宽度有关，而与具体某种编程语言中的具体数据类型无关。例如，在 C 语言中，不管是 char* 类型的指针，还是 int* 类型的指针，还是某个自定义的结构体类型的指针，这些指针的宽度完全相同。在 32 位平台上，指针宽度一定是 32 位；在 64 位平台上，指针宽度一定是 64 位。

因此，JVM 在为 Java 函数计算入参所需要的堆栈空间时，只需要入参的数量即可。

JVM 计算入参堆栈空间的指令如下：

```
__ movptr(rcx, parameter_size);
__ shlptr(rcx, Interpreter::logStackElementSize);
```

这两行代码最终会生成如下机器指令：

```
movl 0x20(%ebp), %ecx
shl $0x2, %ecx
```

movl 0x20(%ebp), %ecx 这条指令的含义是，将 ebp 栈基地址往高地址方向偏移 32 位处的数据（也即 parameterSize 变量的值）传送到 ecx 寄存器中。注意：0x20 是十六进制的写法，换算成十进制就是 32。ecx 是 CPU 中的一个普通寄存器，可以被用于保存临时变量。

接着执行 shl \$0x2, %ecx 这条指令，这条指令的含义是，将 ecx 寄存器中的值左移 2 位。对于二进制数据，左移 N 位，换算成十进制，就是将所对应的十进制数乘以 2 的 N 次方，因此左移 2 位就表示将 ecx 寄存器中的数值乘以 4。为何要乘以 4？因为在 32 位平台上，每一个入参指针都占用 32 位内存，4 字节。0x20(%ebp)处的数据是 parameter_size，每一个 parameter 指针占用 4 字节，因此最终要将其乘以 4。

`shl $0x2, %ecx` 所对应的 C 代码是 `__ shlptr(rcx, Interpreter::logStackElementSize)`, `logStackElementSize` 定义在 `globalDefinitions.hpp` 文件中, 定义如下:

```
#ifdef _LP64
const int LogBytesPerWord = 3;
#else
const int LogBytesPerWord = 2;
#endif
```

`logStackElementSize` 兼容了 32 位和 64 位平台, 如果是 64 位平台, 值是 3, 否则是 2。最终的效果是, 如果在 64 位平台上, 就将 Java 函数入参数量左移 3 位, 相当于乘以 8, 这表示每一个入参都占用 8 字节堆栈空间。同理, 如果在 32 位平台上, 就将 Java 函数入参数量左移 2 位, 相当于乘以 4。

`CallStub()` (即 `stubGenerator_x86_32 : generate_call_stub()` 函数) 执行完 `__ movptr(rcx, parameter_size)` 和 `__ shlptr(rcx, Interpreter::logStackElementSize)` 之后, 就计算出即将被调用的 Java 函数入参所需要的堆栈空间。但是, `CallStub()` 还要保存调用者的数据段现场, 这些用于保存调用者所执行到的 Java 程序所对应的机器指令的基址和变址, 因此 `CallStub()` 接着会执行下面这行代码:

```
__ addptr(rcx, locals_count_in_bytes);
```

这主要用于保存 `rdi`、`rsi`、`rbx`、`mxcsr` 这 4 个寄存器的值。这行代码最终会被翻译成下面的机器指令:

```
add $0x10, %ecx
```

在 32 位平台上, 由于这 4 个寄存器各占 4 字节的内存, 因此需要再将 `ecx` 寄存器加上 16, 最终 JVM 为将被调用的 Java 函数所分配的堆栈空间会再增加 16 字节大小。

基址和变址用于 Java 字节码取指, 一个 Java 函数对应若干条字节码指令, 而一条 JVM 字节码指令由若干机器指令组成 (准确地说, 是转换为若干机器指令), 物理机器能够自动取指, 但是无法对 JVM 字节码进行自动取指, 因此对 JVM 字节码的取指机制需要由 JVM 自己去实现。后面会对此进行详细分析。

上面的指令旨在计算出将被调用的 Java 函数入参所占用的堆栈空间, 可以看到, 最终所占用的空间大小为:

$$\text{Java 函数入参数量} \times 4 + 4 \times 4$$

4×4 就是最后 `rdi`、`rsi`、`rbx`、`mxcsr` 这 4 个寄存器所占用的堆栈空间大小。

完成了 Java 函数入参空间计算后, 接下来就需要执行最主要的一步: 动态分配堆栈内存。这一步很简单, 直接执行 `sub operand, %esp` 即可实现, `operand` 就表示刚才计算出来的堆栈大小。

在 CallStub() (即 stubGenerator_x86_32:generate_call_stub()函数)中使用下面的 C 代码实现:

```
__ subptr(rsp, rcx);
```

这行代码最终会生成下面这条机器指令:

```
sub %ecx, %esp
```

在这条指令之前, JVM 所计算出的堆栈空间大小保存在 ecx 寄存器中, 因此这里直接将 esp 减去 ecx 寄存器的值, 就完成堆栈空间的分配。

为了加速内存寻址和回收, 物理机器在分配堆栈空间时都会进行内存对齐, JVM 也保留了这一原则, 因此完成堆栈内存分配后, 接着 CallStub()执行下面这行代码:

```
__ andptr(rsp, -(StackAlignmentInBytes));
```

其最终对应的机器指令如下:

```
and $0xffffffff0, %esp
```

堆栈按 16 位对齐, 相当于减去后 4 位的值。如果前面为堆栈空间分配的字节数不够 16 的整数倍, 这里就会减小 esp 寄存器的值, 使其按 16 位对齐。

至此, JVM 完成了动态堆栈内存分配, 这是 JVM 最具里程碑意义的事件!

这一关迈过去之后, JVM 终于跨越 C/C++ 程序与 Java 程序之间的桥梁, 翻越中间的分水岭, 纵身一跃, 开始要进入 Java 程序的领域“地界”中了。进入 Java 的世界, 向着詹爷当年的伟大目标大步前进! 作者每每阅读至此, 总会有种神圣的使命感, 更有一种踏上伟大征程的激情! 这一刻, 完全可以比肩当年法拉第向戴维呈现会议演讲稿, 那个足以改变人类社会和生活的方方面面, 使人类迈向现代化的历史时刻; 完全可以比肩当年爱因斯坦发表相对论, 那个足以推翻经典物理学, 使人类全新认识物理、宇宙、时间并因此而得以进入太空探索的历史时刻。因为这一刻, IT 界即将发生翻天覆地的变革, 若干程序员不用再面对内存、指针、寄存器, 商业编程的门槛极大地降低, 信息化得以飞速发展, 移动互联网得以普及, 并因此带动大数据、人工智能、物联网、云计算等热门领域的深度发展, 因此再次改变人类的消费、交易等社会活动方式, 深刻变革经济政治的内在结构和布局, 使人类向着更加高级的文明进化。虽然肯定不少人会觉得我言过其实, IT 界各种伟大的发明实在太多, 但 Java 的确是站在了巨人的肩上。如同不能因为法拉第和爱因斯坦太过伟大就否认为此做出各种铺垫研究的前人的惊人成就, 同样不能否认 Java 所依赖的全部重要变革的成就, 但是的确要承认, Java 的出现推动了很多领域的高速发展。

十万个感叹号, 向詹爷致敬!!!

5. CallStub：调用者保存

JVM 为即将被调用的 Java 方法分配了堆栈空间，调用者是 CallStub()所指向的函数，其实就是 stubGenerator_x86_32:generate_call_stub()。接下来 JVM 就要将 CPU 的控制权转交给被调用的 Java 方法，但是在转交之前，调用者需要保存自己的寄存器数据，这些寄存器主要包括：edi、esi、edx。

这里稍微交代下关于汇编的部分背景知识，没兴趣的小伙伴大可略过不看，不影响后续理解。懂汇编的朋友都知道，内存中一切数据都是二进制，不管是“真的”数据，还是机器指令，都是数据。如果不加以区分，你可以将一个机器指令当做一串普通的数字，也可以将一个数据看成是一个特定的机器指令。计算机区分内存中的一块数据到底是机器指令，还是普通的数据，主要取决于 CS:IP 寄存器，被 CS:IP 寄存器所指的内存数据就是机器指令，会被 CPU 执行，否则就是数据，CPU 可以对其进行数据传送。

每次在执行函数调用时，CS:IP 寄存器会从当前调用者函数的机器指令处跳转到被调用函数，这样 CPU 才能执行被调用的函数。但是当被调用函数执行完了后，CPU 需要跳转到调用者函数中继续执行调用者函数的机器指令，换言之，需要恢复 CS:IP 的值，使之重新指向调用者函数中执行被调用函数的下一条指令。如何恢复呢？前文讲过，每次发生函数调用时，机器会将调用者函数的 CS:IP 压入栈中，而在函数调用结束后，再次将调用者函数的 CS:IP 从栈中弹出来进行恢复。

物理机器通过 CS:IP 来区分一个内存中的数据到底是真实的数据还是机器指令，而对于数据，物理机器一般会用 edi 和 esi 分别保存目的偏移地址和源偏移地址。例如在字符串复制时，一段优化的汇编代码中会同时使用 edi 和 esi，分别指向目标字符串索引位置和源字符串索引位置。

而在 JVM 中，edi 和 esi 却被赋予了更多神圣的职责，例如在 Java 函数调用过程中，esi 会用于 Java 字节码寻址。每当 JVM 开始执行 Java 函数的某个字节码指令时，JVM 会首先将 esi 寄存器指向目标字节码指令的偏移地址，然后 JVM 跳转到该字节码所对应的第一个机器指令开始执行。

所以 edi 和 esi 在 JVM 中是与调用者函数紧密关联的寄存器，是调用者函数的私有数据。

ebx 是一个通用的寄存器，但是也经常用来作为一段数据的基地址，例如使用汇编对一个一维数组的成员元素进行寻址，可以将 ebx 定位到这个一维数组的起始地址，然后使用一个变址定位到数组中的某个元素。在 JVM 中，ebx 便被赋予了这种非常实际的作用，在执行 Java 函数调用时，ebx 会用来存放 Java 函数中即将被执行的字节码指令的基地址，然后通过 jmp 指令跳转到该字节码位置进行字节码解释执行。因此 ebx 也会与 edi、esi 一样，与调用者函数息

息相关，也是调用者函数的私有数据。

既然 esi、edi、ebx 都属于调用者函数的私有数据，因此在发生函数调用之前，调用者函数必须将这些数据保存起来，因为在被调用者函数中，这些数据也会被被调用函数所使用，其中的数据会被被调用函数修改，这样，当被调用函数执行完毕，程序流重新回到调用者函数中时，如果调用函数之前没有保存这些数据，这些数据就无法恢复，从而使程序发生异常。

esi、edi 和 ebx 的保存并不是必须的，例如随便写一段 C 程序然后编译，编译器往往并不会保存这些数据，因为很多编译器只会使用有限的寄存器保存函数特有的数据，而不会使用 esi、edi 和 ebx 这 3 类寄存器。

保存的方式有很多种，可以将其保存到应用程序的堆中，也可以保存到栈中。由于 esi、edi 和 ebx 可以被看做是调用者函数的私有数据，因此 JVM 直接将其保存到了被调用者函数的堆栈中。注意，是保存到了被调用函数的堆栈中，而不是调用函数的堆栈中。关于这一点，只是一种约定俗成的做法，其实保存到调用者函数的堆栈中也是可以的。无论保存到谁的堆栈中，只要在被调用函数执行完之后系统能够恢复调用者函数的这些私有数据即可。

保存 esi、edi、ebx 很简单，如下：

清单：/src/cpu/x86/vm/stubGenerator_x86_32.cpp

作用：generate_call_stub()函数

```
address generate_call_stub(address& return_address) {
    //...

    // save rdi, rsi, & rbx, according to C calling conventions
    //保存 rdi
    __ movptr(saved_rdi, rdi);
    //保存 rsi
    __ movptr(saved_rsi, rsi);
    //保存 rbx
    __ movptr(saved_rbx, rbx);

    //...
}
```

这几条模板最终会生成如下机器指令（使用汇编助记符表示）：

```
mov    %edi,-0x4(%ebp)
mov    %esi,-0x8(%ebp)
mov    %ebx,-0xc(%ebp)
```

这 3 条指令执行之后，堆栈布局如图 2.24 所示。

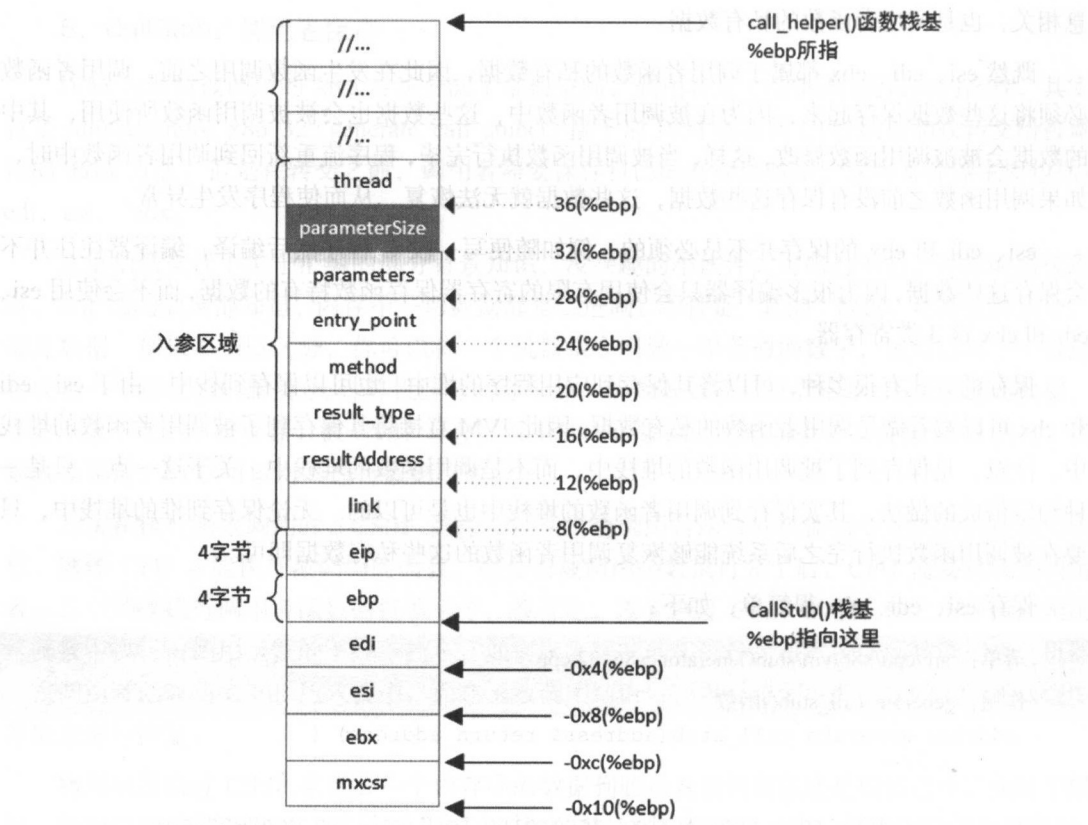


图 2.24 堆栈布局

注意，JVM 还保存了 `mxcsr` 寄存器，这属于 Intel 的 SSE 技术，该议题比较高级，如果要详细讲的话估计会占去不少篇幅，有兴趣的小伙伴可以自行研究，这里就不“歪楼”了。

以上过程有一个专门的术语，叫作“现场保存”。当调用者函数的现场全部保存完之后，CPU 的控制权马上就要移交给被调用者函数了。

6. CallStub：参数压栈

前面在分析“动态分配堆栈”时，分析出 `CallStub` 函数指针为即将调用的函数分配的堆栈空间大小为：

$$\text{Java 函数入参数量} \times 4 + 4 \times 4$$

4×4 就是最后 `rdi`、`rsi`、`rbx`、`mxcsr` 这 4 个寄存器所占用的堆栈空间大小。

`CallStub` 为被调用者函数所分配的堆栈空间大小完全取决于 Java 函数的入参数量，为了分

析内存空间分布情况，这里假设即将被调用的 Java 函数包含 3 个参数，则执行完前面两步（分别是动态分配堆栈和保存调用者现场）之后，堆栈实际布局如图 2.25 所示。

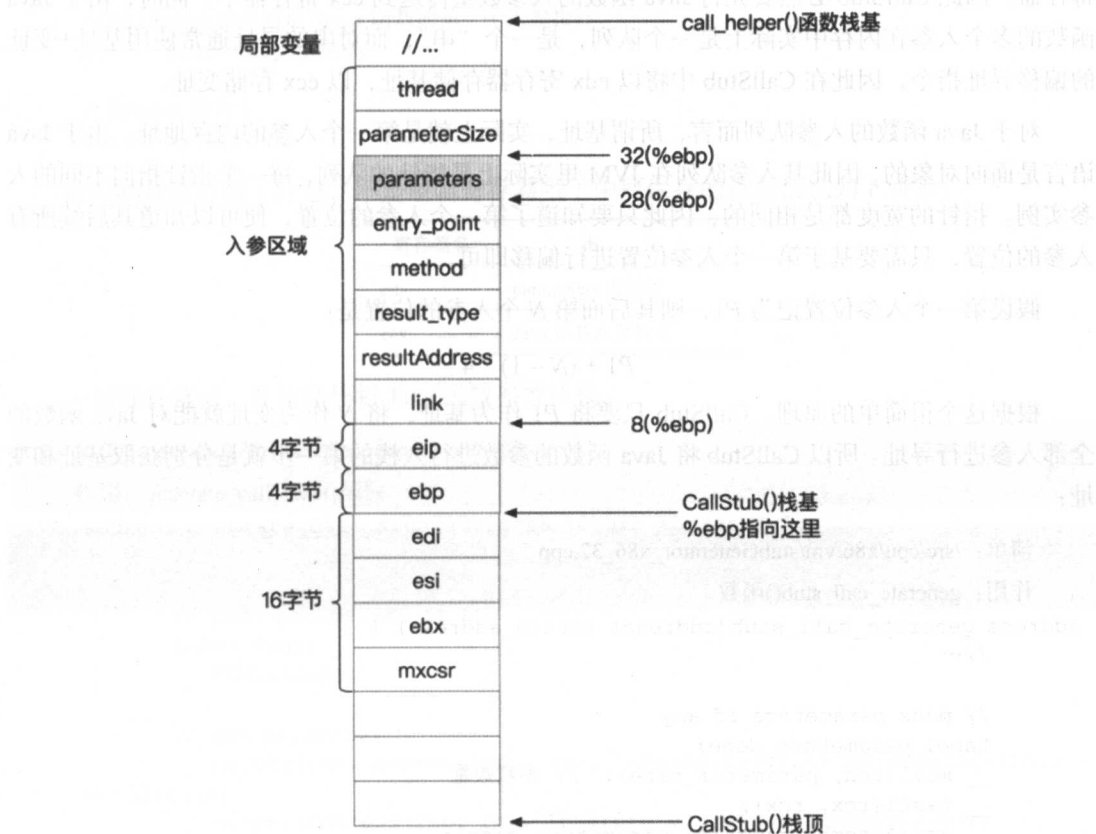


图 2.25 CallStub 分配堆栈和保存调用者现场后的内存布局

CallStub 为调用者分配的堆栈空间还剩余 3 个数据需要填充，接下来 JVM 要做的就是将即将被调用的 Java 函数的入参复制到这剩余的堆栈空间里去。

既然要进行数据复制，CallStub 至少要知道两点：

- ◎ 即将被调用的 Java 函数的入参数量有多少。
- ◎ 即将被复制的 Java 函数的参数集合在哪。

这两个要素在进入本流程之前，全都已经计算得到，并且作为 CallStub 的参数传递给了 CallStub 所指向的函数。在图 2.25 中，28(%ebp)代表的堆栈位置保存的便是 Java 函数的入参数量，而 32(%ebp)所代表的堆栈位置保存的便是 Java 函数的第一个入参。

由于不同的 Java 函数的入参数量是不同的，因此 CallStub 使用了循环进行处理，而这种循环直接是基于机器指令的。在机器层面进行循环，一个约定俗成的做法是将循环次数暂存到 ecx 寄存器。因此 CallStub 必然要先将 Java 函数的入参数量传送到 ecx 寄存器中。同时，由于 Java 函数的多个入参在内存中实际上是一个队列，是一个“串”，而对串的寻址通常使用基址+变址的偏移寻址指令，因此在 CallStub 中将以 edx 寄存器存储基址，以 ecx 存储变址。

对于 Java 函数的入参队列而言，所谓基址，实际上就是第一个入参的内存地址。由于 Java 语言是面向对象的，因此其入参队列在 JVM 里实际上是指针的队列，每一个指针指向不同的入参实例。指针的宽度都是相同的，因此只要知道了第一个入参的位置，便可以知道其后续所有入参的位置，只需要基于第一个入参位置进行偏移即可。

假设第一个入参位置记为 $P1$ ，则其后面第 N 个入参的位置是：

$$P1 + (N - 1) * 4$$

根据这个很简单的原理，CallStub 只要将 $P1$ 作为基址，将 N 作为变址就能对 Java 函数的全部入参进行寻址。所以 CallStub 将 Java 函数的参数进行入栈的第一步就是分别获取基址和变址：

清单：/src/cpu/x86/vm/stubGenerator_x86_32.cpp

作用：generate_call_stub()函数

```
address generate_call_stub(address& return_address) {
    //...

    // pass parameters if any
    Label parameters_done;
    __ movl(rcx, parameter_size); // 参数数量
    __ testl(rcx, rcx);
    __ jcc(Assembler::zero, parameters_done);

    // parameter passing loop
    __ movptr(rdx, parameters); // 第一个入参地址
    __ xorptr(rbx, rbx);

    //...
}
```

这段模板在 JVM 启动过程中会生成如下机器指令（使用汇编助记符给出）：

```
//将 32(%ebp) 处的数据传送给 ecx 寄存器
//32(%ebp) 保存的是 Java 函数入参数量，即 parameter_size
mov    0x20(%ebp), %ecx
```

```
//校验 parameter_size 是否为 0，若是 0 则直接跳过参数处理
```

```
test    %ecx,%ecx
je      0xb370b68b;;该地址每次启动 JVM 时都不一样

//将 28(%ebp) 处的数据传送给 edx 寄存器
//28(%ebp) 保存的是 Java 函数的第一个入参指针
mov     0x1c(%ebp),%edx

//把%ebx 设为 0
xor     %ebx,%ebx
```

此时物理寄存器（注意，不是逻辑寄存器哦）中所保存的重要信息如下所示：

寄存器名	指 向
edx	parameters 首地址
ecx	Java 函数入参数量

一切准备就绪，开始循环将 Java 函数参数压栈：

```
清单：/src/cpu/x86/vm/stubGenerator_x86_32.cpp
作用：generate_call_stub()函数

address generate_call_stub(address& return_address) {
    //...

    // pass parameters if any
    Label loop;
    __ BIND(loop);

    // get parameter
    __ movptr(rax, Address(rdx, rcx, Interpreter::stackElementScale(), -
wordSize));
    __ movptr(Address(rsp, rbx, Interpreter::stackElementScale(),
    Interpreter::expr_offset_in_bytes(0)), rax);    //
store parameter
    __ increment(rbx);
    __ decrement(rcx);
    __ jcc(Assembler::notZero, loop);

    //...
}
```

这段模板最终会生成下面的机器指令（使用汇编助记符表示）：

```
mov     -0x4(%edx,%ecx,4),%eax
mov     %eax,(%esp,%ebx,4)
inc     %ebx
dec     %ecx
jne     0xb370b696
```


在机器层面进行循环，一般有两种方式：一种是使用 `loop` 指令；另一种则使用跳转。很显然这里使用了跳转。

熟悉汇编的小伙伴可能看出来这段汇编中对循环因子 `ecx` 做的是减法（`dec %ecx`，表示减去 1），而常见的循环中一般是做加法，一般使用 `inc %ecx`。这主要是因为 `CallStub` 对 Java 函数的入参采取的是逆向遍历，也就是从后往前遍历参数，并将读取到的入参传送到堆栈中。

上面这段太“底层”了，很显然这会让没有汇编基础的小伙伴们难以理解，所以还是通过举例子的方式来说明。

假设被调用的 Java 函数包含 3 个入参，分别使用 `arg1`、`arg2` 和 `arg3` 表示，则 `CallStub` 指针所指向的函数的 `parameters` 入参指向 Java 函数实际存储 Java 函数 3 个入参的内存区的首地址。此时堆栈空间布局如图 2.26 所示。

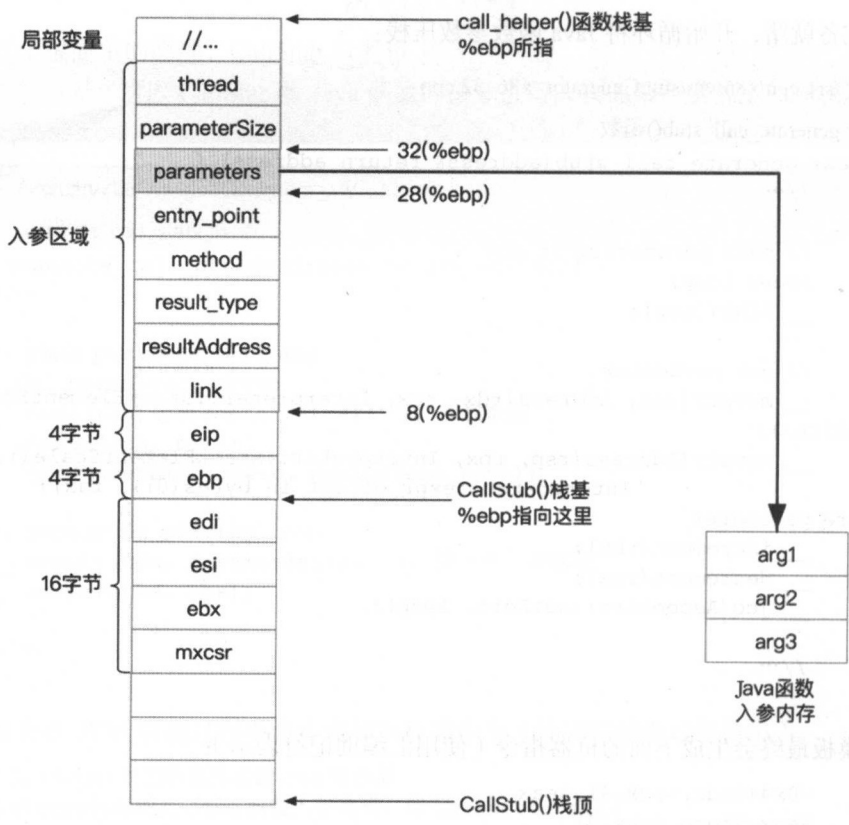


图 2.26 Java 函数参数入栈之前的堆栈布局

当第一轮循环完成之后，Java 函数的第三个入参被压栈，如图 2.27 所示。

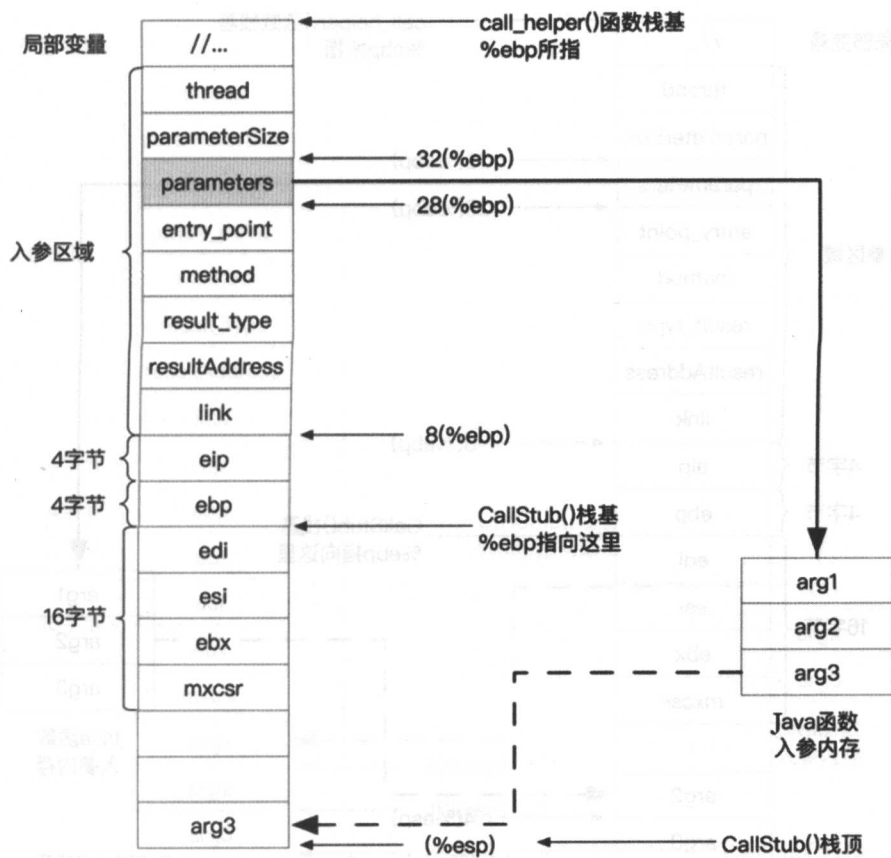


图 2.27 Java 函数参数第一轮压栈

第二轮压栈后，堆栈空间布局如图 2.28 所示。

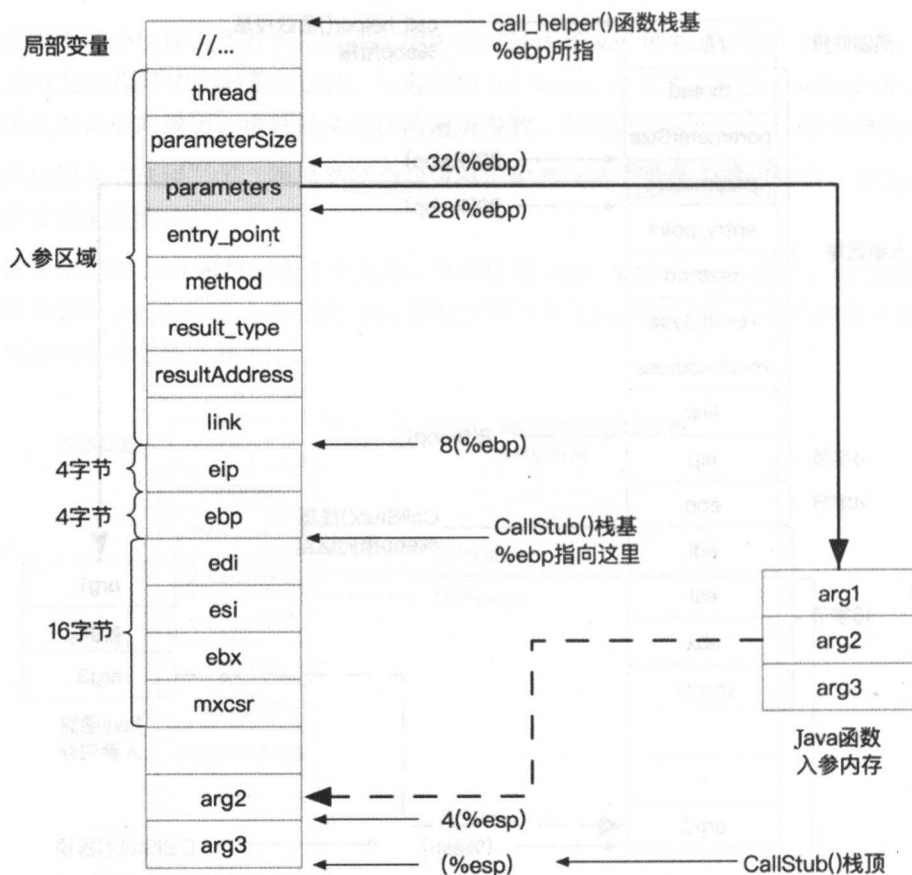


图 2.28 Java 函数参数第二轮压栈

第三轮压栈后，堆栈空间布局如图 2.29 所示。

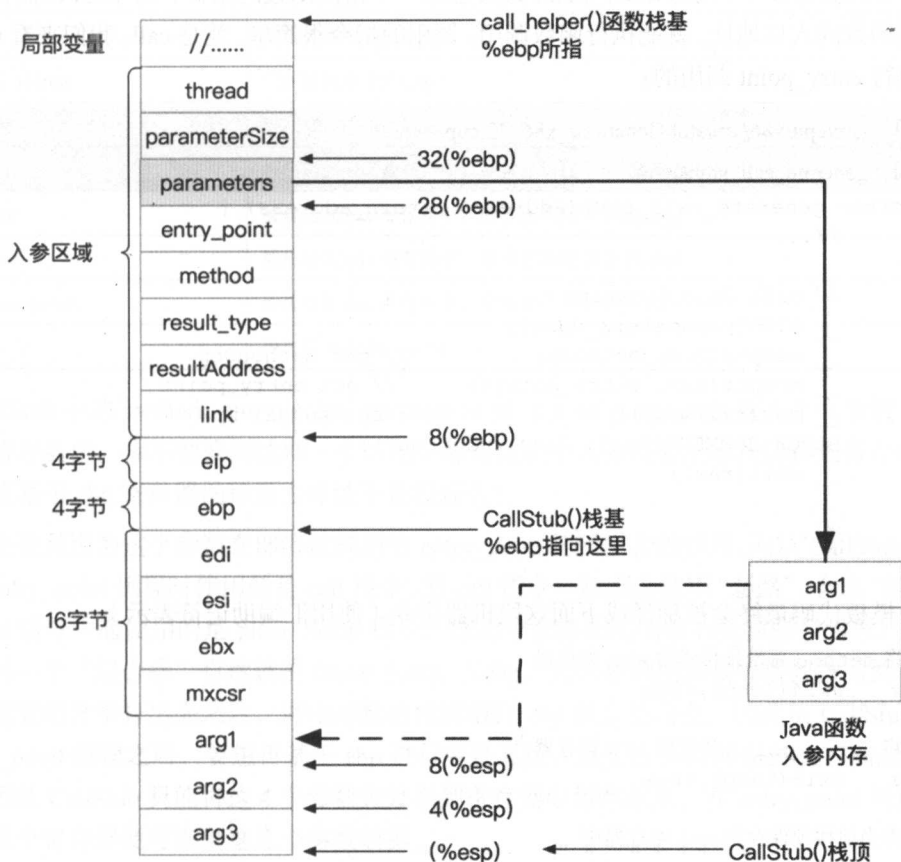


图 2.29 Java 函数参数第三轮压栈

至此，Java 函数的 3 个入参全部被压入栈中。离 Java 函数的调用越来越近了。

7. CallStub：调用 entry_point 例程

前面经过调用者框架栈帧保存（栈基）、堆栈动态扩展、现场保存、Java 函数参数压栈这一系列的逻辑处理，JVM 终于为 Java 函数的调用演完前奏，一切就绪，就等着吹响进攻的冲锋号。而负责吹响冲锋号的，就是 entry_point 例程。

关于到底啥是 entry_point 例程，为何要取这么一个古里古怪的名字，放到后文再解释。这里只需要知道这是一个与 CallStub 一样的例程即可。

在 JVM 调用 CallStub 所指向的函数时，已经将 entry_point 例程的首地址传递给 CallStub 函数了，作为其第 5 个入参。entry_point 其实也是一个指向函数的指针，对于 CPU 而言，只要能够拿到函数的入口地址，就能执行函数调用。调用的指令很简单，就是 call。我们来看 CallStub 是如何执行 entry_point 调用的：

清单：/src/cpu/x86/vm/stubGenerator_x86_32.cpp

作用：generate_call_stub()函数

```
address generate_call_stub(address& return_address) {
    //...

    // call Java function
    __ BIND(parameters_done);
    __ movptr(rbx, method);           // get methodOop
    __ movptr(rax, entry_point);      // get entry_point
    __ mov(rsi, rsp);                 // set sender sp
    BLOCK_COMMENT("call Java function");
    __ call(rax);

    //...
}
```

这段模板代码最终会被翻译成下面这段机器指令（使用汇编助记符表示）：

```
//将 method 首地址传递给 ebx 寄存器
mov    0x14(%ebp),%ebx

//将 entry_point 传递给 eax 寄存器
mov    0x18(%ebp),%eax

//将当前栈顶保存到 esi 寄存器中
mov    %esp,%esi

//调用 entry_point
call   *%eax
```

这段机器指令很简单，主要将 Java 函数所对应的 method 对象的首地址保存到 ebx 寄存器中，同时将 CallStub 栈顶地址保存到 esi 寄存器中。

眼尖的小伙伴们会发现，到这里为止，貌似 CallStub 函数入参的一半都已经被传送到相关寄存器中保存起来了，那么，其他入参的去向如何呢？下面给出了到目前为止 CallStub 函数所有入参的去向一览表（见表 2.5），表中清楚地显示了各入参的现状。

表 2.5 CallStub 函数入参

参 数	保存位置
link	仍在堆栈中 8(%ebp)
result_val_address	仍在堆栈中 12(%ebp)
result_type	仍在堆栈中 16(%ebp)
method	被传送到 ebx 寄存器中，原本在堆栈中 20(%ebp)
entry_point	被传送到 eax 寄存器中，原本在堆栈中 24(%ebp)
parameters	被传送到 edx 寄存器中，原本在堆栈中 28(%ebp)
size_of_parameters	被传送到 ecx 寄存器中，原本存在于堆栈中 32(%ebp)
CHECK	仍在堆栈中 36(%ebp)

CallStub 中的 method、entry_point、parameters 和 size_of_parameters 这 4 个人参被从堆栈中传送到寄存器中。你不禁要问这样一个问题，为何这几个人参需要从堆栈转移到寄存器中呢？如果一直基于 ebp 寄存器偏移量去寻址不是很好么？

这主要是因为这个参数在即将被调用的 entry_point 例程中会被使用，而从 CallStub 例程“跳转”到 entry_point 例程时使用的是 call 指令，而 call 指令一般都会使用“套路”，会出“组合拳”，配合 call 指令一起使用的是 push %ebp 指令，该指令会将调用函数的栈帧保存起来。配合 call 指令的另一个“组合拳”自然就是 move %esp, %ebp，其将被调用函数的栈帧指向调用函数的栈顶。这套组合拳打出去之后，调用函数的栈帧指针 ebp 就会被改变，因此从 CallStub 例程进入 entry_point 例程之后，要想再基于 ebp 进行变址寻址，就无法获取到 method、parameters 等参数，因此 CallStub 只能将这 4 个参数先复制到寄存器中暂存起来，在 entry_point 例程中通过读取这几个寄存器便可恢复这几个参数数据。

不过又有一个问题，为何 CallStub 的另外 4 个参数不用通过寄存器临时存储起来呢？这是因为另外 4 个参数在 entry_point 例程中不会被使用到，entry_point 例程不关心这几个数据，只有 CallStub 例程关心。CallStub 调用完 entry_point 例程再回到 CallStub 例程后，ebp 又重新指向 CallStub 例程的栈帧，因此通过 ebp 栈帧进行变址选址，依然能够获取这 4 个参数。

在 CallStub 执行 call %eax 指令之前，物理寄存器（注，不是逻辑寄存器哦）中所保存的重要信息如表 2.6 所示。

表 2.6 物理寄存器信息

寄存器名	指 向
edx	parameters 首地址
ecx	Java 函数入参数量

续表

寄存器名	指 向
ebx	指向 Java 函数，即 Java 函数所对应的 method 对象
esi	CallStub 栈顶
eax	entry_point 例程入口

此时 `eax` 寄存器已经指向了 `entry_point` 例程入口，因此 `CallStub` 只需执行 `call %eax` 指令便可以
可以直接跳转到 `entry_point` 例程，去执行 `entry_point` 例程。

在 `entry_point`，还会经过一段“铺垫”性的逻辑处理，并最终寻找到 Java 函数的第一个字节码
节码并从第一个字节码开始执行。

在前面的叙述中，多次提到 `CallStub` 的一个入参——`method` 对象，该对象代表即将被调用的
Java 函数，通过该对象可以寻址到 Java 函数所对应的第一个字节码指令。那么这个对象到底是啥，
在 `entry_point` 中究竟如何使用呢？

要理清这些问题，不得不先研究清楚 JVM 的另一个重要的主题——内存模型。将 JVM 的内存模型
理清之后，`method` 对象的结构、Java 类的内存结构等概念都会被逐一破解，这也是理解 `entry_point`
例程所必须掌握的基础。没有这个基础，研究 `entry_point` 无异于盲人摸象。

8. CallStub: 获取返回值

`CallStub` 执行 `entry_point` 例程调用时，使用的是 `call` 指令，而非 `jmp`，因此最终 `entry_point`
例程执行完毕之后，CPU 的控制权还是会回到 `CallStub`，继续执行 `entry_point` 例程调用之后的
指令。

调用完 `entry_point` 例程之后，会有返回值，`CallStub` 会获取返回值并继续处理。

本章前面详细描述了物理机器执行 `call` 调用的原理，简而言之就是，物理 CPU 执行 `call` 调用
时，会将 `ip` 压入栈中。`ip` 是一种专门的段寄存器，通常与 `cs` 段寄存器一起用于指向下一条
即将被 CPU 执行的指令地址，这样当被调用函数执行完成之后，CPU 只需要从栈顶取出 `ip`，
便能重新定位到调用函数的下一条指令地址，继续执行调用函数中的逻辑。

当 `CallStub` 执行完 `call %eax` 这条指令后，堆栈内存的布局如图 2.30 所示（假设目标 Java
函数包含 3 个入参）。

在 JVM 内部，调用函数被压栈的 `ip` 寄存器值有一个专门的称谓——`return address`，即返回
地址。注意，返回地址与“返回值”是两个不同的概念，返回值是指被调用函数所返回的结果
值，而返回地址则是指调用函数执行被调用函数所对应指令的下一条指令的内存地址。在下一
章讲解 `entry_point` 时还会对此进行细述。

理解清楚这一点，在下一章就好理解 entry_point 例程中的“return address”这个参数了。事实上，在图 2.30 所示这种堆栈内存布局中，最下面的 eip 在 entry_point 中的描述统一变成了“return address”，因此在下一章所绘制的堆栈内存布局图中，这个存储单元的名称便也统一改成“return address”，诸君谨记！在后续流程中，JVM 为了让被调用函数的入参、局部变量、固定帧以及操作数栈在内存上相连，会不断移动 eip 在堆栈中的位置，各位道友只需要知道，在后续章节中所描述的 return address 这个东东，就是紧跟在 call *%eax 后面的那条指令的地址。

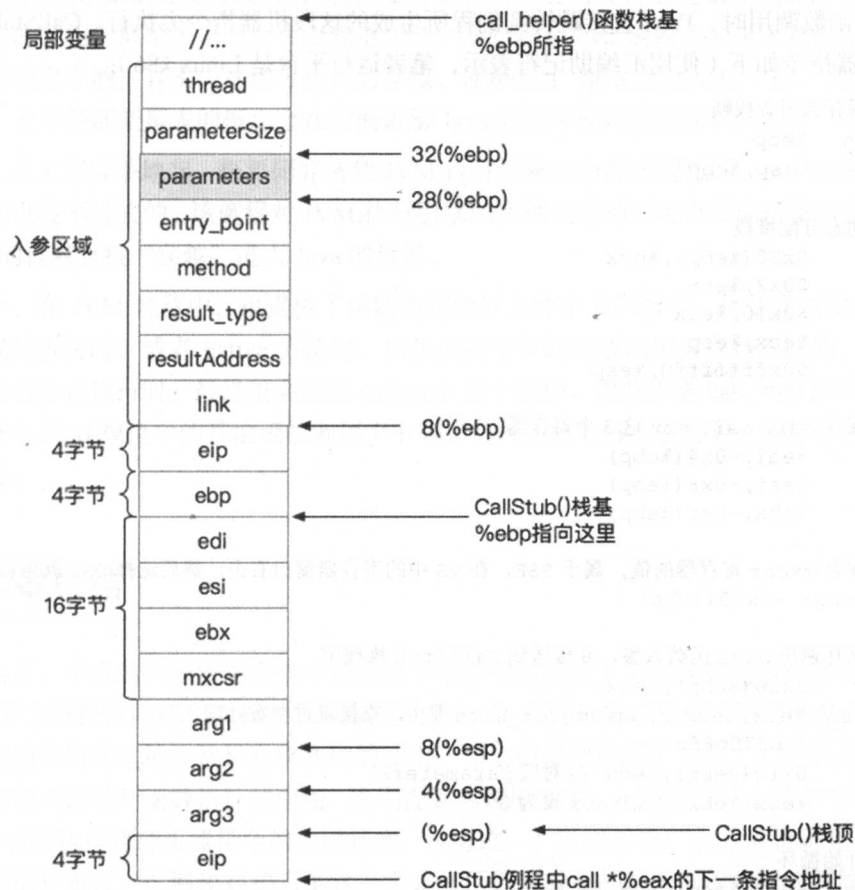


图 2.30 CallStub 执行完 `call *%eax` 指令后的内存布局图

CallStub 中接下来的两条指令如下：

```

◎ mov    0xc(%ebp),%edi //result
◎ mov    0x10(%ebp),%esi //result_type

```

这 2 条指令分别读取被调用函数所返回的值 `result` 和数据类型 `result_type`。看看图 2.30 所示的内存布局图就知道了，`0xc(%ebp)`和`0x10(%ebp)`这 2 个堆栈位置所保存的正是 `resultAddress` 与 `result_type`。JVM 将这 2 个值分别存储进 `edi` 和 `esi` 这 2 个寄存器中，调用方在获取被调用函数的返回值与返回类型时，也会从这 2 个寄存器中读取，这种完全是基于约定的技术实现方式。

9. CallStub: 汇编指令总览

所谓例程，就是一段预先写好的函数，JVM 通过例程函数在启动过程中生成机器指令，当执行 Java 函数调用时，JVM 直接跳转到例程所生成的这段机器指令去执行。CallStub 例程最终生成的机器指令如下（使用汇编助记符表示，笔者运行平台是 Linux x86）：

```
//保存调用者栈帧
push    %ebp
mov     %esp,%ebp

//动态分配堆栈
mov     0x20(%ebp),%ecx
shl     $0x2,%ecx
add     $0x10,%ecx
sub     %ecx,%esp
and     $0xffffffff0,%esp

//保存 edi、esi、ebx 这 3 个寄存器
mov     %edi,-0x4(%ebp)
mov     %esi,-0x8(%ebp)
mov     %ebx,-0xc(%ebp)

//保存 mxcsr 寄存器的值，属于 SSE，在 VS 中的寄存器窗口右击，然后选择 SSE 就可以看到了
stmxcsr -0x10(%ebp)

//循环遍历 Java 函数入参，并传送到 CallStub 堆栈中
mov     0x20(%ebp),%ecx
test    %ecx,%ecx //parameter_size 是 0，直接跳过参数处理
je      0xb370b68b
mov     0x1c(%ebp),%edx //对应 parameters
xor     %ebx,%ebx //把%ebx 设为 0

//开始循环
mov     -0x4(%edx,%ecx,4),%eax
mov     %eax,(%esp,%ebx,4)
inc     %ebx
dec     %ecx
jne     0xb370b696

//开始 entry_point 例程调用
mov     0x14(%ebp),%ebx //对应 method
```

```

mov    0x18(%ebp),%eax //对应 entry_point
mov    %esp,%esi
call   *%eax

```

```

// call_stub_return_address:

```

```

mov    0xc(%ebp),%edi //result
mov    0x10(%ebp),%esi //result_type

```

call_stub 例程的讲解至此先告一段落。到目前为止，各位道友仍然没有看到 JVM 内部如何从 C/C++ 程序完成 Java 函数的调用，因为这部分逻辑需要在后文讲解完 entry_point 例程后才能完全揭开它的真相，但是大家至少可以知道 JVM 是如何将 Java 函数入参压入到机器层面意义上的方法堆栈之中的。其实，这部分压入的参数，便形成了 Java 方法堆栈中的“局部变量表”的一部分，关于局部变量表的概念会在后面讲解 Java 方法堆栈时进行详解。

总之，万丈高楼平地起，想要研究清楚 JVM 执行引擎的内部实现机制，call_stub 这个例程是无论如何也绕不过去的，该例程对 JVM 执行引擎的实现也起到至关重要的桥梁作用，让程序流从 JVM 的世界直接“穿越”进入 Java 的世界。

事实上，在 JVM 规范中，也提供了函数调用的好几种字节码指令，例如在调用 Java 静态方法和类成员方法时，或者 native 方法时，所生成的字节码函数调用指令是不同的，其中部分函数调用的指令在执行时，最终也会经过 call_stub 这个例程，因此弄懂 call_stub 例程的实现机制，对于研究 JVM 规范中的其他函数调用字节码指令的执行原理大有好处，能够启迪思维，然后触类旁通。

2.6 本章总结

总体而言，本章的内容放在全书中都是属于难度非常高的（后面还有两章的难度非常高，即 JVM 的字节码执行原理和 Java 方法栈帧这两章），涉及到大量汇编和 C 语言的基础知识，尤其是涉及到函数指针这种非常具有挑战性的技术，相信很多即使做了多年 C/C++ 开发的底层开发者可能都很少接触到函数指针的应用，更别提其实现原理了。而函数指针正是 JVM 内部实现 C/C++ 程序直接执行物理机器指令的关键技术，没有之一！所以对于广大非常想研究 JVM 执行引擎机制的道友而言，汇编和 C 语言的这一关是必须要闯过去的，否则你的梦想可能永远都实现不了。

本章主要讲解了 JVM 内部的 call_stub 例程的定义和调用机制，但是并没有一开始便直入主题，主要是考虑到技术的难度太大，很多人理解不了。因此作者只能用心良苦地设计了很多 C 和汇编程序，从物理机器执行函数调用的机制开始讲起，为大家揭开物理机器在调用函数时

涉及的若干细节。

接着抛出问题：若想要在 C/C++ 程序中直接执行本地机器指令，技术上该如何实现呢？答案是函数指针。只有通过函数指针才能实现这一目标。而 JVM 内部正是通过 `call_stub` 这个函数指针实现从 JVM 的世界穿越进 Java 的世界。

理解了 `call_stub` 函数指针的设计背景之后，本章接着详细分析了 `call_stub` 函数指针的声明、定义和对应例程的实现机制，花了大量篇幅讲解 `call_stub` 例程的 `pc()` 函数、入参定义、现场保存、堆栈空间动态分配、参数压栈以及对 `entry_point` 例程的调用。不仅详细分析了这些关键步骤的技术实现，还阐述了为何需要这样实现。可以这么说，Java 语言本身的设计宗旨（跨平台和面向对象）决定了 JVM 内部必须要能够由 C/C++ 程序直接执行本地机器指令，而这种技术的实现又决定了 `call_stub` 内部必须这么实现，别无他法。

其实，JVM 与其他任何一个成功的人或物一样，都是满满的套路，套路很深哟！

第 3 章

Java 数据结构与面向对象

本章摘要

- ◎ 数据结构是什么，为什么需要数据结构
- ◎ 数据结构的发展历史以及与算法的关系
- ◎ Java 数据结构的实现机制
- ◎ Java 数据结构——面向对象之技术必然性与偶然性
- ◎ Java 数据结构的字节码格式分析
- ◎ 大端与小端

上一章讲解了 JVM 内部的 `call_stub` 函数指针及其所对应的例程逻辑，难度比较大，属于口味很重的“菜系”，所以本章就来点味道淡一点的“菜”，以免大家消化不良。荤素搭配，咸淡相宜，才能吃得有滋有味。

本章开始聊一聊关于数据结构和算法的那些事儿。我们知道，算法和数据结构是计算机的基础，而 Java 虚拟机的执行引擎框架仍然以此为基础。如同物理 CPU 能够识别字节码和存储单元，C 语言编译器能够识别使用 C 定义的结构体一样，JVM 执行引擎同样能够识别 Java 语言中的基础数据结构——类型，所以 Java 语言里的数据类型对于 JVM 执行引擎而言非常重要。我能想到的最浪漫的事儿，就是看着 JVM 执行引擎对 Java 类型和对象说：“我认得你，我们好像在哪儿见过，你记得吗？”

笔者试图通过历史上的一些事件，来推测詹爷当年将 Java 语言设计成面向对象语言的背景和原因，试图分析 Java 语言如此设计在技术上的偶然性和必然性。本章内容总体上比较“轻松”，大家可以当作小说去阅读。

3.1 从 Java 算法到数据结构

如果非要用一句简单的话来概括何谓“编程”，那么下面的这句话虽然不一定能够反映“编程”的全貌，但至少能够从一个侧面描述编程的本质：

“编程就是使用合适的算法处理特定的数据结构。”

同理，也可以使用下面这句话来概括什么是“程序”：

“程序就是算法与数据结构的有机结合。”

注：这里所说的算法是一种广义的概念，凡是能够完成有特定逻辑的一组指令的集合都可以称为算法。使用 C# 完成一个绚丽的 PC 版视窗是一种算法，使用 C++ 完成一个基于 epoll 的 IO 框架是一种算法，使用 VB 为 Excel 写一个宏脚本是一种算法，使用 Java 完成了一个网页数据加载也是一种算法。这里的算法不局限于那些特定的用于解决数学问题的技术逻辑。

算法是指令驱动的，一条条指令按照一定顺序执行，彼此协作，最终实现某种特定的逻辑，便完成了特定的算法。Java 程序的算法由 Java 字节码指令所驱动。而数据结构往往会作为算法的输入、输出和中间产出，即使输出的是一个简单的整型数字，也是一种数据结构。

本来设计算法是一件挺快乐的事儿，当使用 JavaScript 完成了一个能够绘制柱状统计图的小组件的时候，内心一定是激动无比的；或者当使用汇编完成了一个哈希表设计的时候，一定会觉得自个儿特厉害。但是当算法遇上了不同的操作系统，或者不同的硬件平台，一件原本很快乐的事情便硬生生地被变成一件痛苦的事儿。例如，你想开发一款高并发、高性能的网络通信组件，或者一种分布式的调度器，如果选择的不是 Java 或者 Python，而是 C、C++ 等语言，那么多线程的处理、网络接口的调用等与平台和硬件相关的算法一定会让你如火的热情降到冰点以下。

詹爷之所以伟大就在于，他借助于 JVM 虚拟机和中间语言字节码，完成了指令的跨平台兼容，从而使得 Java 算法能够兼容大部分主流平台。这直接导致算法又变成了一项有趣的活儿，开发者终于又可以集中精力于编写算法这一件事上面，再也不用担心自己的算法是否兼容其他平台，也不用关注底层不同的实现。如同耕耘一样，没有耕坏的地，只有累死的牛，换言之，没有兼容不了的平台，只有写不出的程序。

当年詹爷实现 Java “write once, run anywhere” 的终极武器是“Java 字节码”这种中间语言（ML）技术。这条技术路线的选择有其偶然性，但是更多的却是一种必然性。

虽然可以选择的技术实现方式有很多种，但是总体思路是被限制死的。总而言之，在目前

计算机硬件执行架构的限制下，可选的“技术路径”只有那么几条，总结起来只有以下3条。

1) 直接编写目标硬件平台的机器码指令

这种方式最直接，也最有效，目标硬件平台所支持的指令该是什么就是什么，没有任何二义性，不会产生任何混淆，清清楚楚，明明白白。不过一般只有最底层的软件程序才需要使用这种方式，同时早期的软件程序也采用这种方式，毕竟那时候的编程语言还没有如今这么智能化。

2) 使用高级语言编程，通过编译器实现兼容

可以这么说，如果在马路上拿起一块小石头随便往大街上扔过去，砸中的十有八九就是这种机制的高级编程语言，例如 C、C++、Delphi 等都是如此。这种机制要求比较高，不仅要求在不同的目标平台上安装对应的编译器，还需要软件程序本身针对不同的目标平台调用不同的底层 API。从源代码，到编译器，到打包，再到编译后的目标文件乃至可执行程序，全部都是目标平台相关的，没有一个是只“write once”就能实现“run anywhere”的，需要多个不同平台版本的编译器，源程序也是如此，因此开发者比较痛苦。

3) 通过虚拟机实现兼容性

这种方式就是以 Java 为代表的技术路线。使用这种机制实现兼容性的成本相当低，开发者只需要开发一次源代码，只需要在某一种硬件平台上编译通过，然后便可以在任何目标平台上运行（说任何平台确实夸张了，事实上 JVM 也并没有通吃全部的平台，但是基本实现了几大主流硬件平台和操作系统的兼容性），开发者所需要做的仅仅是在不同的硬件平台上部署不同的虚拟机而已。这种成本相比诸如 C、C++之类的高级编程语言动不动就要针对不同目标平台而修改底层 API，已经几乎可以忽略不计，如果非要为它定一个市场价的话，那一定是跳楼价了。

当然，除了这3种主要的方式，还有其他各种与此类似的实现方案，例如 Python 作为一门脚本语言，不需要编译便可以直接运行，但是其在本质上还是与 JVM 虚拟机类似，因为 Python 的编译过程由解释器代劳，解释器将 python 编译为中间语言（类似于 Java 的字节码）并基于中间语言实现跨平台，在这一点上，与 JVM 完全保持一致。

比较这3种方式，可以发现，实现兼容性的思路基本是从原始笨拙向着高度智能的方向发展，最终达到不需要为了兼容性而额外写多份源代码的目标，即“write once”。

“write once”这一目标的实现，带来的回报或者红利是十分丰厚的（现实社会也是这样，一个良好的顶层社会结构设计，往往会带来社会全方面的飞速发展，并还会额外馈赠很多东西）。“首当其冲”的红利就是开发者再也不用感知底层 API 了，省去了这么个麻烦并且痛苦的过程，

直接使得类似于 Java 这样非常高级的编程语言的学习门槛大幅度降低，甚至近乎于零，虽然这可能会导致开发者在底层实现原理领域存在盲区，不利于技术成长，但是对于商业项目而言，屏蔽了底层实现的高级编程语言的开发效率会成倍地加快。相比之下，在宏观层面上有这么一门近乎智能的编程语言，的确加速了信息化的进程，至于底层硬件的那些东西就交给社会大分工去调节吧，总会有一部分人会专注于那一块像原始森林一样深厚广袤的领域，比如笔者便一直专注于底层的研究。

“write once”所馈赠的另一个回报就是——数据结构不再直接依赖于物理机器。程序是算法与数据结构的有机结合体，不管是算法还是数据结构，最终都需要被物理机器所理解。任何一门编程语言，构成算法基础的指令都会被还原成机器指令，只有物理机器才有能力执行各种各样的算法指令。虽然广义的数据结构是一种抽象的概念，甚至很多时候人们为了演算某种算法，会将抽象的数据结构具象化，使用形象化的语言符号来描述这种抽象的数据结构，但是在工程实践中，数据结构必然需要由一种具体的编程语言去实现。这种实现的背后，仍然是物理机器在支撑，仍然离不开机器指令。简单到定义一个字节变量，复杂到定义一个复合型的结构体，其实都是需要 CPU 首先能够识别对应的机器指令然后才能吭哧吭哧运行实现的。

詹爷在编程领域算得上是一个比较有品味的人，当年设计 Java 时，对这种全新的编程语言期望很高，不仅要求能够实现跨平台兼容，还要求融入当时非常时髦的设计思想——面向对象。Java 语言里面除了接口和枚举，其他的数据结构都是类型（事实上对于 JVM 而言，接口和枚举也是类型），或者反过来说，Java 中的类型都是一种专门的数据结构，整个 Java 程序都由数据结构所组成，Java 算法也由数据结构的动作所驱动，所以数据结构可以说是 Java 的核心。

詹爷为了实现算法“write once, run anywhere”的伟大目标，最终选择了使用字节码中间语言这条技术路线，通过 Java 字节码来统一描述算法。在“字节码”这条技术的康庄大道上，詹爷一溜烟跑到头，一竿子插到底，詹爷将字节码的思想贯彻得非常彻底，不仅将程序算法“字节码”化，连同数据结构一起被“字节码”化。这种“字节码”化的数据结构，由于以“面向对象”为设计宗旨，因此在面向用户（即开发者）的一端是十分人性化的，同时由于被“字节码”化，因此在面向机器的那端，是十分不可理喻的，因为字节码化的数据结构不认识机器，机器也不认识它，这便实现了数据结构对物理机器的去依赖。

总而言之，詹爷使出了浑身解数，不仅让算法变成一种充满趣味的活儿，解除了算法对物理机器指令的依赖，让天下没有难以兼容的平台，而且还让数据结构也脱离对底层硬件平台的依赖，使广大开发者受益匪浅，可以像定义抽象的数据结构那样，随心所欲任意组装所需要的结构体。算法与数据结构对物理硬件的双重去依赖，使得 Java 程序具有简单易学的特点，并最终保证了 Java 程序跨平台兼容的彻底性。

詹爷让数据结构脱离底层机器约束的思路其实并不复杂，例如下面这个 Java 类，是一个描

述 iPhone 6s 手机参数的数据结构:

清单: Iphone6s.java

作用: 使用 Java 类描述 iPhone 6S 手机参数

```
public class Iphone6s {
    int length = 138;        //长度 (毫米)
    int width = 67;         //宽度 (毫米)
    int height = 7;         //高度 (毫米)
    int weight = 143;       //重量 (克)
    int ram = 2;            //ram容量 (G)
    int rom = 16;           //rom容量 (G)
    int pixel = 1200;       //摄像头像素 (万)
}
```

编译这个可以被看成是一个纯粹的“数据结构”的 Java 类, 得到的字节码格式的“数据结构”信息如下:

清单: Iphone6s.java

作用: 使用 Java 类描述 iPhone 6S 手机参数

Compiled from "Iphone6s.java"

```
public class Iphone6s extends java.lang.Object
```

```
    SourceFile: "Iphone6s.java"
```

```
    minor version: 0
```

```
    major version: 50
```

```
    Constant pool:
```

```
const #1 = class      #2;    // Iphone6s
const #2 = Asciz      Iphone6s;
const #3 = class      #4;    // java/lang/Object
const #4 = Asciz      java/lang/Object;
const #5 = Asciz      length;
const #6 = Asciz      I;
const #7 = Asciz      width;
const #8 = Asciz      height;
const #9 = Asciz      weight;
const #10 = Asciz     ram;
const #11 = Asciz     rom;
const #12 = Asciz     pixel;
const #13 = Asciz     <init>;
const #14 = Asciz     ()V;
const #15 = Asciz     Code;
const #16 = Method    #3:#17; // java/lang/Object."<init>":()V
const #17 = NameAndType #13:#14; // "<init>":()V
const #18 = Field      #1.#19; // Iphone6s.length:I
const #19 = NameAndType #5:#6; // length:I
const #20 = Field      #1.#21; // Iphone6s.width:I
const #21 = NameAndType #7:#6; // width:I
```

```

const #22 = Field      #1.#23; // Iphone6s.height:I
const #23 = NameAndType #8:#6; // height:I
const #24 = Field      #1.#25; // Iphone6s.weight:I
const #25 = NameAndType #9:#6; // weight:I
const #26 = Field      #1.#27; // Iphone6s.ram:I
const #27 = NameAndType #10:#6; // ram:I
const #28 = Field      #1.#29; // Iphone6s.rom:I
const #29 = NameAndType #11:#6; // rom:I
const #30 = Field      #1.#31; // Iphone6s.pixel:I
const #31 = NameAndType #12:#6; // pixel:I
const #32 = Asciz      LineNumberTable;
const #33 = Asciz      LocalVariableTable;
const #34 = Asciz      this;
const #35 = Asciz      LIphone6s;;
const #36 = Asciz      SourceFile;
const #37 = Asciz      Iphone6s.java;

```

编译后所得到的这样一种使用字节码进行格式化的数据结构，你很难轻易地将其与具体的机器指令关联起来，而事实上这些“字节码”化的指令也的确无法被物理机器直接识别，这便实现了 Java 数据结构对物理机器的去依赖。不过这种字节码格式终归仍然需要被物理机器理解并执行，这就需要将字节码格式最终转换为最底层的机器指令，这种转换的机制将在后续章节详细分析。

詹爷使尽了浑身解数折腾出 Java 这么一款面向对象的语言，从算法与数据结构的角度看，也是具有重要意义的。其巨大作用便在于，让程序员可以专注于业务逻辑，而不需要将很多精力“浪费”在各种底层平台的接口兼容上。这对于商业项目而言，无疑具有无比巨大的实实在在的价值。

3.2 数据类型简史

前面讲过，虽然 Java 的类型信息通过字节码进行了格式化，但是这种编译后的格式化了的数据类型并不能直接被物理机器识别，没有哪台物理机器能够在读取了 Java 字节码文件内容之后就能直接在内存中构建出对应的结构体。Java 的这种数据结构实现机制相比于同时代的其他编程语言很独特。

数据结构与物理机器之间有着千丝万缕的联系。程序算法告诉物理机器应该怎么做，而数据结构则告诉物理机器拿什么去做。而事实上物理机器的大部分指令也的确同时包含了“怎么做”和“做什么”这两部分。例如下面这段汇编程序：

```

subl$32, %esp
movl$138, -28(%ebp)
movl$16, -8(%ebp)

```

在这段汇编程序中，出现了 2 个熟悉的身影——sub 和 mov（如果前面章节的内容你认真看过的话）这两个指令分别表示减和传送。这两个指令都是典型的带有“立即数”的机器指令，指令本身告诉物理机器要“怎么做”，而跟在指令后面的立即数则告诉物理机器“做什么”。例如对于 subl\$32, %esp 这条指令而言，sub 是物理机器指令，告诉机器要开始做减法运算了，减多少呢？后面的立即数\$32 就是答案。对谁做减法运算呢？再后面的%esp 寄存器就是答案。

在这个例子里，立即数 32 和寄存器 esp 都可以认为是一种数据类型，只不过这是最简单的数据类型，简单到它就是它“自己”，而不包含任何其他元素或成员，这种情况下，数据类型已经直接退化成了数据。一般而言，所谓数据结构，至少也应该是复合结构，但是在物理机器层面，已经没有“复合”这种概念了，CPU 只能操纵位或者存储单位，哪怕再稍微复杂一点点都不能支持。

在这里，有一个关键的点需要明确，那就是数据结构与数据类型的关系。简单地说，数据结构的实现需要依赖数据类型的支持，例如使用 C 语言定义一个单向链表，往往需要多种数据类型的参与才能实现。任何一种编程语言，简单到最原始的机器和汇编，复杂到现代的高级语言，不管哪一种，其实都能够实现任何一种复杂的数据结构，道理很简单，任何编程语言所定义的任意复杂的数据结构最终都要依靠机器指令才能实现，这本身便说明机器指令能够实现任意复杂的数据结构。区别在于实现的难易程度，编程语言所能支持的数据类型越多，则实现复杂的数据结构的成本往往越小。由于机器指令和汇编语言不支持多样化的和复杂的数据类型，因此给程序设计带来了诸多困扰，软件设计师虽然也能够使用基本数据类型通过在内存中建立映射关系从而实现结构化的内存空间布局以支撑对应的算法逻辑，但是仍然无法使用一种高级和直观的数据视图去形象化地表现复杂的对象。例如，构建一种简单的结构体来描述 iPhone 6S 信息，仅仅使用汇编这种并不支持任何数据类型的编程语言也能够实现，但是其在语法层面并不能够提供简单的支持，只能写成下面这种方式：

清单：iphone6s.s

作用：使用汇编为 iPhone 6S 定义“数据结构”

```

main:
    pushl    %ebp
    movl    %esp, %ebp
    subl    $32, %esp
    movl    $138, -28(%ebp)    //长度（毫米）
    movl    $67, -24(%ebp)    //宽度（毫米）
    movl    $7, -20(%ebp)    //高度（毫米）
    movl    $143, -16(%ebp)    //重量（克）

```

```
movl$2, -12(%ebp)      //ram 容量 (G)
movl$16, -8(%ebp)       //rom 容量 (GB)
movl$1200, -4(%ebp)     //摄像头像素 (万)

movl$0, %eax
leave

ret
```

可以看出，这种方式毫无结构感可言，如果不是编写程序的人特意说明这段程序是在定义一个关于 iPhone 6S 的数据结构，读者压根儿就看不出来。

在计算机技术问世之初，其就像太阳一样，散发出万丈光芒，继 18 世纪末 19 世纪初伟大的工业革命变革之后，再一次照亮了全世界，给人类带来了希望和信心。可是，当人们意识到无法随心所欲地定义清晰明了的数据结构的问题之后，发现整个天空都暗了。其时软件领域的天空上漂浮着一朵乌云，这朵乌云叫作“数据结构化”。如果这朵乌云始终不被驱散，那么整个软件界都会处于一片黑暗中，找不着出路。

这个问题不解决，虽然各种各样降低时间复杂度和提升性能算法依然会陆续登上历史舞台，但是没有一种直观的数据结构的视图化编程方式，一切都按照机器的“意志”行事，那么所谓的“数据结构”便只是一种仅仅存在于观念和意识中的概念，看不见摸不着写不出读不懂，始终不能落地，就像一个如幽灵般捉摸不定的量子。因此算法的发明速度也会迟缓，而离开了算法的软件技术，也必定举步维艰得不到真正的发展。可以说，软件在短短几十年里得到迅速发展，与高级编程语言在语法层面提供的自定义数据类型的能力是息息相关的，相关的佐证俯拾皆是，并且这种相关性在 Java 语言里尤其明显，例如 Java 直接内建了 `hashmap`、`list`、`set`、`stack` 等高级数据结构类型，这些类型在大部分 Java 工程中被大量使用，而在汇编或者 C 程序中，开发者往往要自己实现这些高级的数据结构与算法，虽然可以使用第三方包，但是很多第三方包的作者并不保证程序的健壮性和兼容性。

所以现代高级编程语言的发展历程是伴随着对数据类型的日益强大的支持的过程。

回到 IT 历史发展的早期阶段，虽然伟大的先辈们从算法层面对汇编进行了抽象，逐渐发展出了高级编程语言，使开发者不用直面机器指令或者寄存器之类的硬件，但是在数据结构层面一直还没有诞生出这种意识，一切算法仍然围绕着二进制位、存储单位（字节）、字、双字等最原始的数据类型。那时候所谓的二进制位、字节甚至都不能算是数据类型，因为当时根本就没有这个概念。

不过乌云终归是浮云，在聪明的前辈们的努力之下，这朵浮云神马都不是，很快便烟消云散了。IT 科学家们拨开云雾，赫然发现“人工智能”这扇大门，并且科学家们很快便找到了打开这扇智慧大门的金钥匙，于是各种蕴含着“人工智能”的高级编程语言纷纷登场，不过其过

程却是异常艰辛。

最早的语言仅支持少量的数据结构，如 Fortran 90 之前通常用数组来模拟链表及二叉树，而所谓数组，其实汇编里面也能实现，本质上而言并不属于高级编程语言所独有，因为直接使用机器指令也能模拟出一个类似于数组的内存空间。

至于 COBOL、ADA 和 PL/I 之类的早期编程语言，也没有好到哪里去，在数据类型的概念上并没有产生质的飞跃。

及至到了 ALGOL，终于在数据结构方面迈出了历史性的一步。ALGOL 开创性地引入了用户自定义类型的概念，注意，是自定义类型哟！虽然 ALGOL 仅提供少数的基本类型以及少量灵活的结构定义操作符，却允许开发者自主设计一种数据结构。显然，这是数据类型发展过程中最重要的进步。

到了 1967 年，Simula 67 首次提出“类型”的概念，把数据和被允许施与数据上的操作结合为一个统一体，从而成为现代“抽象数据类型”的开端及第一个“面向对象语言”。

通过以上历史进程可见，数据类型并非天生就有，而是在无数前辈们的努力下逐渐提出来的一种概念。从一开始完全没有数据类型的概念，到数据类型概念的萌芽，再到能够自定义数据类型，最终到“抽象数据类型”，其中对数据类型的每一次认识上的加深，都对软件领域历史的发展产生革命性的作用，每一次都极大地提升了编程语言对客观世界的抽象和描述能力，编程语言变得越来越易用，越来越智能，越来越“人性化”。

到 C 语言之父 Richard 发明出 C 语言的时候，人们已经可以轻轻松松地定义和实现数据类型了，已经能够描述非常复杂的客观事物了。例如，同样是为 iPhone 6S 定义一个结构，使用 C 语言的“结构体”可以这样写：

清单：iphone6s.c

作用：使用 C 语言为 iPhone 6S 定义数据结构

```
// 声明一个结构体
struct iphone6s{
    int length;           //长度（毫米）
    int width;            //宽度（毫米）
    int height;           //高度（毫米）
    int weight;           //重量（克）
    int ram;               //ram 容量（GB）
    int rom;              //rom 容量（GB）
    int pixel;            //摄像头像素（万）
};

// 结构体使用
int main(){
```

```
struct iphone6s iphone; //定义变量
```

```
// 结构体初始化
```

```
iphone.length=138;
```

```
iphone.width=67;
```

```
iphone.height=7;
```

```
iphone.weight=143;
```

```
iphone.ram=2;
```

```
iphone.rom=16;
```

```
iphone.pixel=1200;
```

```
return 0;
```

```
}
```

C 语言为了实现“数据结构”的可视化，定义了“结构体”这种类型，通过结构体，开发者可以将任意类型的基本数据组合到一起，形成复杂的数据结构。不仅如此，C 语言的结构体还能嵌套使用，结构体里面包含结构体，从而可以定义出多维度的深度树形结构。例如，将上面的结构体改造一下，变成嵌套结构体：

清单：iphone6s.c

作用：使用 C 语言为 iPhone 6S 定义数据结构

```
struct volume{
    int length;           //长度（毫米）
    int width;            //宽度（毫米）
    int height;           //高度（毫米）
};
```

```
struct basic{
    int weight;           //重量（克）
    int ram;              //ram 容量（GB）
    int rom;              //rom 容量（GB）
    int pixel;            //摄像头像素（万）
};
```

```
// 嵌套的结构体
```

```
struct iphone6s{
    struct volume volume;
    struct basic basic;
};
```

```
int main(){
    struct iphone6s iphone;
```

```
    struct volume volume;
    volume.length=138;
```



```

volume.width=67;
volume.height=7;

struct basic basic;
basic.weight=143;
basic.ram=2;
basic.rom=16;
basic.pixel=1200;

iphone.volume=volume;
iphone.basic=basic;

return 0;
}

```

在本例中，iPhone 6S 变成了一个嵌套的结构体，基于此例，可以定义任意复杂的结构体。由于 C 语言中有了“结构体”这根“如意金箍棒”，想让它变啥它就变啥，世界上几乎没有它描述不了的事物，从此数据结构有了合适的落脚点，看得见摸得着了。软件业由此大步前进。

除了 C 语言，其他算得上“高级”的编程语言，也几乎都能够从语法层面支持“数据结构”的概念，开发者按照给定的格式去定义数据结构，机器一定能够正确识别。例如在 VB 中可以使用下面这种方式去定义数据结构：

清单：test.vb

作用：使用 VB 语言定义数据结构

```

type student
    id as string
    name as String
    age as int
end type

```

其他常见的编程语言诸如 Python、Perl、C++、PHP 等都支持自定义数据结构。

前面讲过，Java 中的“数据结构”被“字节码”格式化了，最终的结果是 Java 的数据结构解除了对物理机器的依赖。但是 C 语言中的数据结构对物理机器是有依赖的，这种依赖在 C 语言源代码被编译后便出现了。在 Linux 平台上，上面那段 iphone6s.c 的 C 语言数据结构被编译后，会直接变成下面这段汇编程序：

清单：iphone6s.s

作用：C 语言数据结构被编译后的汇编程序

```

main:
    pushl    %ebp
    movl    %esp, %ebp
    subl    $32, %esp

```

```
movl$138, -28(%ebp)
movl$67, -24(%ebp)
movl$7, -20(%ebp)
movl$143, -16(%ebp)
movl$2, -12(%ebp)
movl$16, -8(%ebp)
movl$1200, -4(%ebp)

movl$0, %eax
leave
ret
```

可以看到，C 语言中的数据结构被编译后，直接被转换成了对应平台上的机器指令，因此物理机器是可以直接识别并运行的。虽然被转换后，原本 C 语言中结构体数据的类型信息被彻底抹去，被彻底打回最原始的类型，变得不可理解，如果直接阅读编译后的汇编代码，你肯定不知道原来这段程序是一个数据结构体。

因此，C 语言的结构体类型相比汇编，就如同自动挡轿车相对于手动挡。C 程序通过编译器，将人类可理解的结构体这种具象化的高级概念，转换成了底层非结构化的低级概念，这种转换是自动的，是编译器智能分析的结果。虽然在今天看来，这种自动化并不一定算得上“智能化”，因为当今“智能化”这个词已经有了更加复杂的含义，但是在当年那种编译原理尚未完全成熟的年代，能够做到这种自动转换，已属不易！也许再过 30 年，当人工智能与神经网络的原理和技术十分成熟时，人们再回过头来看今天所谓的大数据等概念，也会觉得一点都算不上“智能”和“高级”呢。

如果说 C 语言中的数据结构依赖于物理机器，倒不如说依赖于特定平台上的特定编译器更加合适，而事实上也的确如此，不同硬件平台上的编译器各不相同，这些不同的编译器将同一个 C 语言程序编译成了各自对应的底层指令。

C 语言在编译期实现了数据结构的解释，编译器实现了对 C 程序中的数据类型的识别、解释，并最终翻译成了机器指令可以识别的数据类型。操作系统加载编译后的二进制程序执行，在内存中构建出与源程序中所定义的完全一致的数据结构，这给人一种错觉，以为物理机器能够认识 C 程序中所定义的各种复杂的结构类型。Java 语言则与此完全不同，Java 编译器虽然在编译时也能够准确分析出 Java 的类型信息，但是编译后的 Java 字节码却并不能直接由物理机器执行，因此 Java 语言的类型信息并不完全在编译期维护，而是推迟到了运行期。原理与之相似的编程语言包括同为面向对象编程语言的 SmallTalk、JavaScript、C#等。

知道了数据类型的发展简史，各位朋友真应该为自己生在这样一个伟大的时代而暗自庆幸，因为现代的各种高级编程语言都有很丰富的数据类型可以选择，如果实在不满足要求，还可以

自定义。而在几十年前，这根本就是一件不可想象的事情。尤其对于 Java 程序员而言，Java 语言不仅类型丰富，而且面向对象，用它来编程，真的是一件简单而快乐的事情。

总而言之，高级编程语言分别从算法和数据类型这两个层面对早期编程语言进行改进，改进的最终结果是：在算法层面，早期编程语言需要使用几十甚至几百行指令才能实现的逻辑，现代高级编程语言往往只需要一两行代码便可完成。而在数据类型层面，现代高级编程语言往往直接将各种常见的结构类型集成到 SDK 中提供给开发者使用，并在此基础上允许开发者重写或自定义。而 Java 在这两方面都达到了一定的历史高度甚至巅峰。

3.3 Java 数据结构之偶然性

Java 丰富的数据类型和面向对象的特性，为广大软件开发者能够自由描述各种复杂的客观事物打开方便之门。不过相比于 C/C++ 等面向过程的语言，或者虽然面向对象但是不够彻底的语言，Java 是实实在在地将“面向对象”贯彻到底的语言。这种贯彻所带来的结果就是，如果不将整型、字符型、浮点型等基本数据类型划分为数据结构的一种，那么 Java 语言中的数据结构就只退化成一种——类型。不管一种被描述的事物多么简单，也不管其如何复杂，在 Java 语言中，事物的一切属性都必须被“打包”为 Java 类型，Java 类型已经成为语法层面的强制约束。而在其他语言中，可能还存在诸如结构体之类的概念，在 Java 中统统不存在（Java 中允许定义枚举类型可能已经很开恩了），尤其在 C/C++ 语言中，甚至允许专门为函数指针定义一种特别的类型。

相比于 JVM 执行引擎在技术上狭窄的选择面，Java 的这种数据结构的技术实现其实带有一定的历史偶然性。下面先回顾一下 JVM 执行引擎的技术实现为何是必然的。

1. JVM 执行引擎的必然选择

在讨论 Java 的面向对象思想之前，先让我们再次回顾下 JVM 的执行引擎。

詹爷当初创立 Java 门派，便立志于改变智能家电程序的开发方式，希望能够开发出一门“write once, run anywhere”的语言。因此，Java 语言天生便口含“兼容”的金钥匙，具体如何兼容？假设用 C 语言开发一款程序，为了实现兼容性，我们需要做表 3.1 中所列这些事情

表 3.1 C 程序的兼容性处理方式

开发	不同的平台上，分别开发不同的代码。例如访问线程、读写文件、申请内存等，不同平台、不同操作系统所提供的 API 一定是不同的，因此所调用的接口也不尽相同
编译	得在不同的平台上，安装不同的编译器，然后才能执行不同的编译命令编译程序
打包	在不同的平台上，需要选择适应该平台的软件包和编译器，对程序进行打包
运行	不同的平台上，需要有不同的安装环境，例如 Windows 上需要各种 dll 动态链接库，Linux 上需要各种 so 支持

而如果选择使用 Java 开发一款软件，同样在开发、编译、打包和运行期，所需要做的兼容性措施如表 3.2 所示。

表 3.2 Java 程序的兼容性处理方式

开发	无论在何种平台上，只需开发同一套程序，无需为底层平台的不同而做特殊处理，Java 提供统一的文件访问、内存申请、线程操作、GUI 开发等接口
编译	不同的是，在不同的平台上需要安装不同的 JDK 相同的是，只需执行同一种编译指令，便能编译程序
打包	无论哪种平台，无论是通过命令行执行打包，还是通过 IDE 执行打包，命令行或 IDE 菜单完全一致
运行	只要安装了 JVM 虚拟机，便能运行，不再依赖任何其他 dll 或 so

对于软件开发，生命周期的绝大部分时间都花费在开发上，而 Java 程序只需要编写一次，就能在“任何”平台上运行。由此带来的效率提升不可谓不显著。Java 除了在设计上保持一致性，编译和打包、运行都保持了高度的兼容性和独立性，开发者所要做的只是针对不同的平台，安装适当的 JVM 和 JDK。

詹爷当年为了实现这种能够带来巨大效益的编程语言，在如何执行程序的问题上花费了无数心血，以其深厚的功力，以其对机器语言、汇编、操作系统的深刻理解，最终终于实现了 JVM 执行引擎。前文花了大量篇幅讲解詹爷为什么最终会选择字节码的方式作为中间语言，以及为什么最终会选择将字节码实时翻译成汇编程序这种技术路径。通过前文的分析可知，JVM 做出这种技术上的选择，是综合考虑了各种技术方案的优势和弊端、特点和特性，最后所得出的最优解。

一句话，JVM 选择这种技术路径是必然的，即使换了另一个大师级的人物来开发，最终也一定会设计成现在 JVM 的样子。只是可能 CallStub 这种分水岭不同，或者给出的诸如 entry_point 这样的例程不同，或者采用的 jit 即时编译器的编译算法不同。但是在现有的物理硬件设备不变的前提下，基本不可能翻腾出别的浪花。这种必然性是各种技术、各种路径碰撞的必然结果，所谓“万宗归一”，不仅在武侠、宗教世界中存在这种终极的“道”，在软件编程领域，也同样存在这样无形的“天道”。

但是，天道不可畏，并非不能突破。每一次的技术革命，都是人类突破自我、超越天道的勇敢尝试。所以，研究 JVM，表面看似似乎并没有什么用，但是，通过窥探 JVM 的本质，无形之中便获得了一种“道”，这种道的能力会使你如龙之遨游四海，如鹰之翱翔天空，这种道能够给你一种智慧，一种举一反三的能力，会使你研究其他领域的速度更快，领悟力更高。举个不恰当的例子，如果将来某一天，计算机硬件得到革命性的创新，那时候计算机不仅能识别 0 和 1，而且能够识别 1000、10000 了，到了那一天，如果你掌握了现在的 0 和 1 理论，那么你就可以根据 0 和 1 时代的经验，在 1000、10000 的基础上构建一系列汇编语言、操作系统、高级编程语言，以及像 JVM 这种能够跨越多个底层平台和硬件的虚拟机。这就是道的力量。

2. Java 面向对象之技术偶然性

如果说 JVM 为了兼容各种异构硬件和平台而作出的这种技术抉择是一种必然，那么 Java 选择面向对象的编程方式和内存管理模型（数据结构总是与内存管理机制联系在一起）便具有一定的偶然性和随机性了。对于给定的执行引擎，可以使用若干种编程方式来使用这种执行引擎。所以，Java 的面向对象机制与 JVM 的执行引擎并不是强绑定的关系，这如同运载火箭一样，不同的卫星可以使用同一种运载工具发射升空，同一种卫星也可以使用不同的运载工具发射升空。Java 程序经编译后的字节码文件，既可以使用 HotSpot 执行，也可以使用 jrockit 解释，或者被 IBM JVM 运行。而随着 JVM 越来越强大、稳定和高效，JVM 本身也成为一种平台性产品，诸多编程语言可以被编译成 JVM 字节码文件，因此，对于同样一个 Java 程序编译后生成的字节码中间文件，你可以选择使用不同的执行引擎来运行，而同一款 JVM 执行引擎，可以解释 Java、PHP 等源码被编译后生成的字节码。

所以，当年詹爷选择使用编译解释的方式来执行 Java 程序，但是其实 Java 语言原本可以不面向对象的，它可以选择面向过程，或者面向函数，或者其他。但是，在那个年代，面向对象的编程方式正如日中天，如火如荼，C++、Java、VB、C#，还有一些脚本语言，例如，JavaScript、Python、Perl、PHP、Delphi 等也是面向对象的。

退一万步讲，即使 Java 语言因为追随潮流，选择了面向对象的编程方式，也完全可以不选择现在的内存模型和垃圾回收机制。但是由于，Java 要求能够在运行时通过反射获取到类型信息，因此最终 Java “被迫”选择了现在的这种内存模型。

C++/Delphi 等编程语言也具有面向对象的特性，但是到了运行期已经完全消除了类型概念，并且也无法在运行期“反射”到类型的成员变量、方法等信息，但是 Java 可以。但是 Java 为什么可以做到在运行期动态反射到类型的成员变量和方法信息呢？说白了也很简单，在 JVM 加载 Java 类的时候，就将 Java 类的类元信息（类元信息即变量和方法）保存到内存中，这样在运行期直接读取目标内存中的数据便能获取到相应的信息。这种类元信息，其实就是一种打包好的

数据结构模板，并且在运行期可被识别。而相比于 C++/Delphi 等同样是面向对象的语言，类模板信息早在编译期便被完全擦除，而 Java 语言在编译后仍然保留了类型的内部结构信息，并且类型结构信息被带到了运行期。

不过话说回来，能够做到在运行期动态反射出类型结构信息，并不能成为编程语言选择拥有面向对象特性的必然理由，更不能因此就非要实现自动内存管理（自动垃圾回收），例如，C++这种面向对象的编程语言通过 RTTI（运行时类型识别）也能够做到在运行期识别类型，但是 C++的内存仍然需要开发者自己去释放。从这个角度看，Java 的自动内存管理机制就显得并不是必须的。然而 Java 选择具备运行时类型识别的特性本身便从一个十分隐晦的层面制约了 Java 必须选择成为一门面向对象的编程语言，为何？类型本身就是一种“闭包”的技术手段，只有先从语法层面实现了“闭包”，才能实现“对象”的概念，否则，何来的属性、成员变量、类方法一说？类型是实现将若干属性和动作打包成为一个整体对象进行统一识别的策略。如果 Java 像 C++那样，类型不作为属性和方法封装的唯一手段，开发者可以随心所欲地在类的外面定义变量和函数，那么对于这部分数据的“运行时识别”必然是一个难题，可能需要通过类似 namespace 或者 filename 这样的机制去实现动态反射了，但是这种反射想想都让人头大，不容易啊！换了是你，你会怎么从技术上让其落地和实现呢？

因此，从运行期动态反射的角度看，Java 语言选择成为一门彻底的面向对象语言，绝对是偶然中的必然。相对来说，Java 的自动内存管理（垃圾回收）机制就带有一种随机性。不过，这也不一定正确。当一门编程语言实现了完全的闭包语法策略（使用类型包装可以认为是闭包的一种），便自然而然具备了自动内存管理的技术基础，或者说实现自动内存管理更加容易。所以闭包便成为很多具备自动内存回收特性的编程语言的语法基础，例如 GO 语言、Phthon、JavaScript 等，虽然大家具体实现闭包的手段不同，但是殊途同归，都是为了能够让虚拟机在自动回收内存时尽量简单。

总体而言，Java 的面向对象和自动内存管理的特性实现，因为要实现运行时类型识别的目标，因此在偶然中带有必然性，而必然中又孕育出偶然性因素。

3.4 Java 类型识别

生活在现代世界的 Java 程序员是快乐的。Java 程序员编写的任何类，Java 虚拟机都能够在运行期识别出来，这实在是让人激动。那么 Java 虚拟机究竟是如何做到这一点的呢？一切奥秘都隐藏在 Java 源程序被编译后生成的字节码文件中。Java 类在编译期生成的字节码有其特定的组织规律，Java 虚拟机在加载类时，对编译期生成的字节码信息按照固定的格式进行解析，一

一步步解析出字节码中所存储的类型结构信息，从而在运行期完全还原出原始的 Java 类的全部结构。

3.4.1 class 字节码概述

每一个 Java 类被编译后会生成一个对应的.class 字节码文件，要想研究 JVM 加载 Java 类的原理，首先必须熟练掌握 Java 类被编译成的.class 格式文件结构。下面将从几个方面来描述字节码的组成格式。

1. class 文件构成基础

在 class 字节码文件中，数据都是以二进制流的形式存储。这些字节流之间都严格按照规定的顺序排列，字节之间不存在任何空隙，对于超过 8 位的数据，将按照 Big-Endian（大端）的顺序存储，即高位字节存储在低的地址上面，而低位字节存储到高地址上面。其实这也是 class 文件跨平台的关键，因为 PowerPC 架构的处理器采用 Big-Endian 的存储顺序，而 x86 系列的处理器则采用 Little-Endian（小端）的存储顺序，因此为了 class 文件在各种异构处理器架构下能够保持统一的存储顺序，虚拟机必须设置统一的存储规范。

2. class 文件的 10 个组成结构

class 字节码文件是采用类似 C 语言的结构体来存储数据的，主要有两类数据项：无符号数和表。无符号数用来表述数字、索引引用以及字符串等，在.class 文件中主要使用的无符号数包括 u1、u2、u4 和 u8，分别代表 1 字节、2 字节、4 字节和 8 字节的无符号数。而表是由多个无符号数以及其他的表组成的复合结构。

一个 class 字节码文件主要由以下 10 部分组成：

- ◎ MagicNumber
- ◎ Version
- ◎ Constant_pool
- ◎ Access_flag
- ◎ This_class
- ◎ Super_class
- ◎ Interfaces
- ◎ Fields
- ◎ Methods

◎ Attributes

这些数据的类型和长度都是不同的，用一个数据结构可以表示如下：

```
ClassFile {
    u4 magic;
    u2 minor_version;
    u2 major_version;
    u2 constant_pool_count;
    cp_info constant_pool[constant_pool_count-1];
    u2 access_flags;
    u2 this_class;
    u2 super_class;
    u2 interfaces_count;
    u2 interfaces[interfaces_count];
    u2 fields_count;
    field_info fields[fields_count];
    u2 methods_count;
    method_info methods[methods_count];
    u2 attributes_count;
    attribute_info attributes[attributes_count];
}
```

要注意的是，在 JVM 内部其实并没有定义这样的一种数据结构，这里使用类似 C 语言中的结构体方式来表示 Java 字节码文件的结构，完全是为了展示不同属性的数据类型和长度。

3. class 文件中的各组成字段简单说明

1) MagicNumber

MagicNumber 是用来标志 class 文件的，位于每一个 Java class 文件的最前面 4 个字节，值固定为 0xCAFEBADE。注意，这个是十六进制数值，并非字符串“CAFEBADE”，其对应的二进制数是 11001010 11111110 10111010 10111110B，一共占 32 位比特即 4 字节。虚拟机加载 class 文件时会先检查这 4 字节，如果不是 0xCAFEBADE，则虚拟机拒绝加载该文件，这样就可以防止加载非 class 文件而造成虚拟机崩溃。

2) Version

Version 字段由 2 个长度都为 2 字节的字段组成，分别是 Major Version 和 Minor Version，代表当前 class 文件的主版本号 and 次版本号。随着 Java 技术的不断发展，Java class 会增加一些新的内容来支持 Java 语言的新特性。同时，不同的虚拟机支持的 Java class 文件的版本范围是不同的，所以在加载 class 文件之前可以先看看该 class 文件是否在当前虚拟机的支持范围之内，避免加载不支持的 class 文件。高版本的 JVM 可以加载低版本的 class，但反之就不行。

目前已发布的 Version 包括: 1.1(45)、1.2(46)、1.3(47)、1.4(48)、1.5(49)、1.6(50)、1.7(51)。对于 JDK 1.6 编译出来的 class file, 其版本号是 0x00000032(查看 class file 的第 5~8 这 4 字节, 下文会详述), 转换为十进制数就是 50。

如果使用低版本的 JDK 编译 Java 程序, 然后使用高版本的 JRE 执行 class file, 能够顺利执行。反之, 如果使用高版本的 JDK 编译 Java 程序, 而使用低版本的 JRE 执行 class file, 则 JVM 会抛出类似于 “java.lang.UnsupportedClassVersionError: Unsupported major.minor version 50” 这样的异常。当出现这种异常信息时, 首先查看 JVM 的版本, 查看方法很简单, 进入 jvm\bin\, 执行 Java -version 命令即可。其次是查看 class 的 Version (即编译该 Java 类的 JDK 版本), 查看方法是, 使用二进制文件编辑器打开对应的 class 文件, 根据其开头第 4~8 个字节的二进制值, 换算出对应的十进制数, 即可得到其版本号。

3) 常量池 (Constant pool)

常量池信息从 class 文件的第 9 字节开始。

首先是 2 字节 (即第 9 和第 10 两个字节) 的长度字段 constant_pool_count, 表明常量池包含了多少个常量。

接下来的二进制信息描述 [constant_pool_count-1] 个常量, 常量池里放的是字面常量和符号引用。

字面常量主要包含文本串以及被声明为 final 的常量。符号引用包含类和接口的全限定名、字段的名称和描述符、方法的名称和描述符, 因为 Java 语言在编译的时候没有连接这一步, 所有的引用都是运行时动态加载的, 所以需要把这些引用的信息保存在 class 文件里。

字面常量根据具体的类型分成字符串、整型、长整型、浮点型、双精度浮点型这几种基本类型。

符号引用保存的是引用的全限定名, 所以保存的是字符串。

4) Access_flag

主要保存当前类的访问权限。

5) This_cass

主要保存当前类的全限定名在常量池里的索引。

6) Super_class

主要保存当前类的父类的全限定名在常量池里的索引。

7) Interfaces

主要保存当前类实现的接口列表，包含两部分内容：`interfaces_count` 和 `interfaces[interfaces_count]`。

- ◎ `interfaces_count` 指的是当前类实现的接口数目。

- ◎ `interfaces[]` 是包含 `interfaces_count` 个接口的全局限定名的索引的数组。

8) Fields

主要保存当前类的成员列表，包含两部分的内容：`fields_count` 和 `fields[fields_count]`。

- ◎ `fields_count` 是类变量和实例变量的字段的数量总和。

- ◎ `fields[]` 是包含字段详细信息的列表。

9) Methods

主要保存当前类的方法列表，包含两部分的内容：`methods_count` 和 `methods[methods_count]`。

- ◎ `methods_count` 是该类或者接口显式定义的方法的数量。

- ◎ `method[]` 是包含方法信息的一个详细列表。

10) Attributes

主要保存当前类 `attributes` 列表，包含两部分内容：`attributes_count` 和 `attributes[attributes_count]`。

这些属性在字节码文件中的具体存储方式，下面会通过一个示例来讲解，毕竟概念还是挺抽象的嘛^_^。

3.4.2 魔数与 JVM 内部的 int 类型

由于魔数在字节码文件中占 4 字节，并且其数据值固定不变，一直都是 0xCAFEBABE，因此 JVM 内部使用 `u4` 这种自定义的数据类型存放魔数。`u4` 这种数据类型的定义如下：

```
清单：/src/share/vm/utilities/globalDefinitions.hpp
```

作用：`u4` 类型定义

```
typedef jint u4;
```

`juint` 也是自定义类型（注意，不是 `junit`），但是这种类型是平台相关的，在 `linux` 平台上，`juint` 定义如下：

```
清单：/src/share/vm/utilities/globalDefinitions_gcc.hpp
```

作用：`juint` 类型定义

```
typedef uint32_t juint;
```

uint32_t 仍然是自定义类型，定义如下：

清单：/src/share/vm/utilities/globalDefinitions_gcc.hpp

作用：joint 类型定义

```
#ifndef _UINT32_T
#define _UINT32_T
typedef unsigned int uint32_t;
#endif
```

由此可知，uint32_t 最终所代表的类型是 unsigned int，在 32 位或 64 位系统平台上，unsigned int 都占 4 字节，正好能够存放下魔数信息。所以，JVM 内部的 u4 数据类型能够存放 4 字节。而 JVM 内部除了 u4，还定义了另外 3 种常用的数据类型：

- ◎ u1, 该数据类型占 1 字节，在 Linux 平台上所代表的 C 语言类型是 unsigned char。
- ◎ u2, 该数据类型占 2 字节，在 Linux 平台上所代表的 C 语言类型是 unsigned short。
- ◎ u8, 该数据类型占 8 字节，在 linux 平台上所代表的 C 语言类型是 unsigned long。

这里额外对 uint32_t 之类的数据类型做一个补充说明，按照 POSIX 标准，一般整型对应的 *_t 类型为：

- ◎ 1 字节，uint8_t
- ◎ 2 字节，uint16_t
- ◎ 4 字节，uint32_t
- ◎ 8 字节，uint64_t

这 4 种基本数据类型在遵循 C99 标准的 C 语言中进行了内置定义，因此开发者可以直接使用。这些数据类型都有一个特点，那就是以 _t 结尾，这其实表示这些数据类型并不是什么新的类型，它们只是使用 typedef 为类型起的别名而已。原理虽然很简单，但是作用却很大，比如 C 中没有 bool，于是在一个软件中，一些程序员使用 int，一些程序员使用 short，会比较混乱，最好就是用一个 typedef 来定义，如：

```
typedef char bool;
```

uint8_t 之类的意义也正是如此，是为了让代码能够被更好地维护，并且大家都使用同一套标准。

3.4.3 常量池与 JVM 内部对象模型

常量池是 Java 字节码文件中比较重要的概念，是整个 Java 类的核心所在，因为常量池中记录了一个 Java 类的所有成员变量、成员方法和静态变量与静态方法、构造函数等全部信息，包

括变量名、方法名、访问标识、类型信息等。

JVM 内部定义了一个 C++ 类型 `constantPoolOop` 来记录解析后的常量池信息（本书默认以 Hotspot 1.6 版本的源码作为研究对象，下同。Hotspot 后续版本略有变化，类名有部分修改，但总体上变化不大）。`constantPoolOop` 其实是别名，其原始的类型是 `constantPoolOopDesc`。在 `/src/share/vm/oops/oopsHierarchy.hpp` 中进行了这两种类型的转换：

```
typedef class constantPoolOopDesc* constantPoolOop;
```

而事实上，JVM 内部为了在运行期描述 Java 类的类型信息和内部结构，定义了很多以 `Desc` 结尾的 `oop` 类，详细的定义见 `/src/share/vm/oops/oopsHierarchy.hpp` 文件。这个源码中所定义类，便于实现 Java 的面向对象特性，很重要哦！各位道友可以先进入这个源码感受一下，第一次看肯定没感觉，但是看多了总会“日久生情”，然后才能深入了解。

既然提到了 `oop` 的概念，就不得不提 JVM 内部对 Java 对象的表示模型，这个模型便是著名的“`oop-klass`”模型。

1. oop-klass 模型

Hotspot 虚拟机在内部使用两组类来表示 Java 的类和对象。

- ◎ `oop`(ordinary object pointer)，用来描述对象实例信息。
- ◎ `klass`，用来描述 Java 类，是虚拟机内部 Java 类型结构的对等体。

JVM 内部定义了各种 `oop-klass`，在 JVM 看来，不仅 Java 类是对象，Java 方法也是对象，字节码常量池也是对象，一切皆是对象。JVM 使用不同的 `oop-klass` 模型来表示各种不同的对象。而在技术落地时，这些不同的模型就使用不同的 `oop` 类和 `klass` 类来表示。由于 JVM 使用 C/C++ 编写，因此这些 `oop` 和 `klass` 类便是各种不同的 c++ 类。对于 Java 类型与实例对象，JVM 使用 `instanceOop` 和 `instanceKlass` 这 2 个 C++ 类来表示，这 2 个类后文会逐步分析到，此处略过不表，各位道友可以先打开 HotSpot 的源码进去看下这 2 个类，找找“感觉”。

这里贴出 HotSpot 内部所定义的 `oop` 大全：

清单：`/src/share/vm/oops/oopsHierarchy.hpp`

作用：描述 HotSpot 中的 `oop` 体系

```
typedef class oopDesc* oop;
typedef class instanceOopDesc* instanceOop;
typedef class methodOopDesc* methodOop;
typedef class constMethodOopDesc* constMethodOop;
typedef class methodDataOopDesc* methodDataOop;
typedef class arrayOopDesc* arrayOop;
typedef class objArrayOopDesc* objArrayOop;
```

```

typedef class    typeArrayOopDesc*    typeArrayOop;
typedef class    constantPoolOopDesc*  constantPoolOop;
typedef class    constantPoolCacheOopDesc*  constantPoolCacheOop;
typedef class    klassOopDesc*          klassOop;
typedef class    markOopDesc*           markOop;
typedef class    compiledICHolderOopDesc* compiledICHolderOop;

```

也许是为了简化变量名, JVM 统一将最后的 Desc 去掉, 全部处理成以 Oop 结尾的类型名。例如对于 Java 类中所定义的方法, JVM 使用 methodOop 去描述 Java 方法的全部信息; 对于 Java 类中所定义的引用对象变量, JVM 则使用 objArrayOop 来保存这个引用变量的“全息”信息。

Hotspot 使用 klass 来描述 Java 的类型信息。Hotspot 定义了如下几种类型信息。

清单: /src/share/vm/oops/oopsHierarchy.hpp

作用: 描述 HotSpot 中的类型信息

```

class    Klass;
class    instanceKlass;
class    instanceMirrorKlass;
class    instanceRefKlass;
class    methodKlass;
class    constMethodKlass;
class    methodDataKlass;
class    klassKlass;
class    instanceKlassKlass;
class    arrayKlassKlass;
class    objArrayKlassKlass;
class    typeArrayKlassKlass;
class    arrayKlass;
class    objArrayKlass;
class    typeArrayKlass;
class    constantPoolKlass;
class    constantPoolCacheKlass;
class    compiledICHolderKlass;

```

纵观以上 oop 和 klass 体系的定义, 可以发现, 无论是 oop 还是 klass, 基本都被划分为来分别描述 instance、method、constantMethod、methodData、array、objArray、typeArray、constantPool、constantPoolCache、klass、compiledICHolder 这几种模型, 这几种模型中的每一种都有一个对应的 xxxOopDesc 和对应的 xxxKlass。通俗而言, 这几种模型分别用于描述 Java 类类型和类型指针、Java 方法类型和方法指针、常量池类型及指针、基本数据类型的数组类型及指针、引用类型的数组类型及指针、常量池缓存类型及指针、Java 类实例对象类型及指针。HotSpot 认为使用这几种模型, 便足以勾画 Java 程序的全部: 数据、方法、类型、数组和实例。

那么 oop 到底是啥, 其存在的意义究竟是什么? 其名称已经说得很清楚, 就是普通对象指针。指针指向哪里? 指向 klass 类实例。直接这么说可能比较难以理解, 举个例子, 若 Java 程

序中定义了一个类 ClassA，同时程序中有如下代码：

```
ClassA a = new ClassA();
```

当 Hotspot 执行到这里时，会先将 ClassA 这个类型加载到 perm 区（也叫方法区），然后在 Hotspot 堆中为其实例对象 a 开辟一块内存空间，存放实例数据。在 JVM 加载 ClassA 到 perm 区时，JVM 就会创建一个 instanceKlass，instanceKlass 中保存了 ClassA 这个 Java 类中所定义的一切信息，包括变量、方法、父类、接口、构造函数、属性等，所以 instanceKlass 就是 ClassA 这个 Java 类类型结构的对等体。而 instanceOop 这个“普通对象指针”对象中包含了一个指针，该指针就指向 klassInstance 这个实例。在 JVM 实例化 ClassA 时，JVM 又会创建一个 instanceOop，instanceOop 便是 ClassA 对象实例 a 在内存中的对等体，主要存储 ClassA 实例对象的成员变量。其中，instanceOop 中有一个指针指向 instanceKlass，通过这个指针，JVM 便可以在运行期获取这个类实例对象的类元信息。

2. oopDesc

既然讲到了 oop，就不得不提 JVM 中所有 oop 对象的老祖宗——oopDesc 类。上述列表里的所有 oopDesc，诸如 instanceOopDesc、constantPoolOopDesc、klassOopDesc 等，在 C++ 的继承体系中，最终全都来自顶级的父类——oopDesc（JDK8 中已经没有 oopDesc，换成了别的名子，但是换汤不换药，内部结构并没有什么太大的变化）。Java 的面向对象和运行期反射的能力，便是由 oopDesc 予以体现和支撑。看起来很神秘不是？但是看看其结构，会发现简单得似乎与其所具备的能力有点不相称：

清单：/src/share/vm/oops/ooDesc.hpp

作用：oopDesc 定义

```
class oopDesc {
    friend class VMStructs;
private:
    volatile markOop _mark;
    union _metadata {
        wideKlassOop _klass;
        narrowOop _compressed_klass;
    } _metadata;

    // Fast access to barrier set. Must be initialized.
    static BarrierSet* _bs;

    //...
}
```

抛开友元类 VMStructs，以及用于内存屏障的 _bs，oopDesc 类中只剩下了 2 个成员变量（友

元类并不算成员变量): `_mark` 和 `_metadata`。其中 `_metadata` 是联合结构体, 里面包含两个元素, 分别是 `wideKlassOop` 与 `narrowOop`, 顾名思义, 前者是宽指针, 后者是压缩指针。关于宽指针与窄指针这里先简单提一句, 主要用于 JVM 是否对 Java class 进行压缩, 如果使用了压缩技术, 自然可以节省出一定的宝贵内存空间。

`oopDesc` 的这 2 个成员变量的作用很简单, `_mark` 顾名思义, 似乎是一种标记, 而事实上也的确如此, Java 类在整个生命周期中, 会涉及到线程状态、并发锁、GC 分代信息等内部标识, 这些标识全都打在 `_mark` 变量上。而 `_metadata` 顾名思义也很简单, 用于标识元数据。每一个 Java 类都会包含一定的变量、方法、父类、所实现的接口等信息, 这些均可称为 Java 类的“元数据”, 其实可以更加通俗点, 所谓的元数据就是在前面反复讲的数据结构。Java 类的结构信息在编译期被编译为字节码格式, JVM 则在运行期进一步解析字节码格式, 从字节码二进制流中还原出一个 Java 在源码期间所定义的全部数据结构信息, JVM 需要将解析出来的结果保存到内存中, 以便在运行期进行各种操作, 例如反射, 而 `_metadata` 便起到指针的作用, 指向 Java 类的数据结构被解析后所保存的内存位置。

仍然以上一节所举的实例化 `ClassA` 这个自定义 Java 类的例子进行说明。当 JVM 完成 `ClassA` 类型的实例化之后, 会为该 Java 类创建对应的 `oop-klass` 模型, `oop` 对应的类是 `instanceOop`, `klass` 对应的类是 `instanceKlass`。上一节讲过, `instanceOop` 内部会有一个指针指向 `instanceKlass`, 其实这个指针便是 `oopDesc` 中所定义的 `_metadata`。 `klass` 是 Java 类型的对等体, 而 Java 类型, 便是 Java 编程语言中用于描述客观事物的数据结构, 而数据结构包含一个客观事物的全部属性和行为, 所以叫做“类元”信息, 这便是 `_metadata` 的本意。

`_metadata` 的作用可以参考图 3.1 所示。

JVM 内部一切都是对象, 而描述这些对象的共同祖先就是 `oopDesc`, 但是 `oopDesc` 的结构却异常简单, 这不禁让人感到隐隐不安, 担心如此简单的结构如何能够描述 Java 类复杂的数据结构。这个问题暂且放下不表, 后面会揭示出事实真相。让我们将目光移回到常量池对象。

常量池这种对象虽然并不是由程序员在源代码中定义的, 但是在 Java 类被编译后, 其成为编译后的字节码中最核心的对象, 对于这种重量级的“人物”, JVM 当然也有对应的类型去描述它。上文讲过, JVM 使用 `constantPoolOop` 这种类型来保存常量池的信息, 但是 `constantPoolOop` 内部其实还是借助于 `typeArrayOop` 这种类型才可以对常量池进行描述。 `constantPoolOop` 的结构定义在 `/src/share/vm/oops/constantPoolOop.hpp` 中, 本书后面章节会详细讲解, 这里就先不展开, 大家可以自己进入这个源码文件感受下。

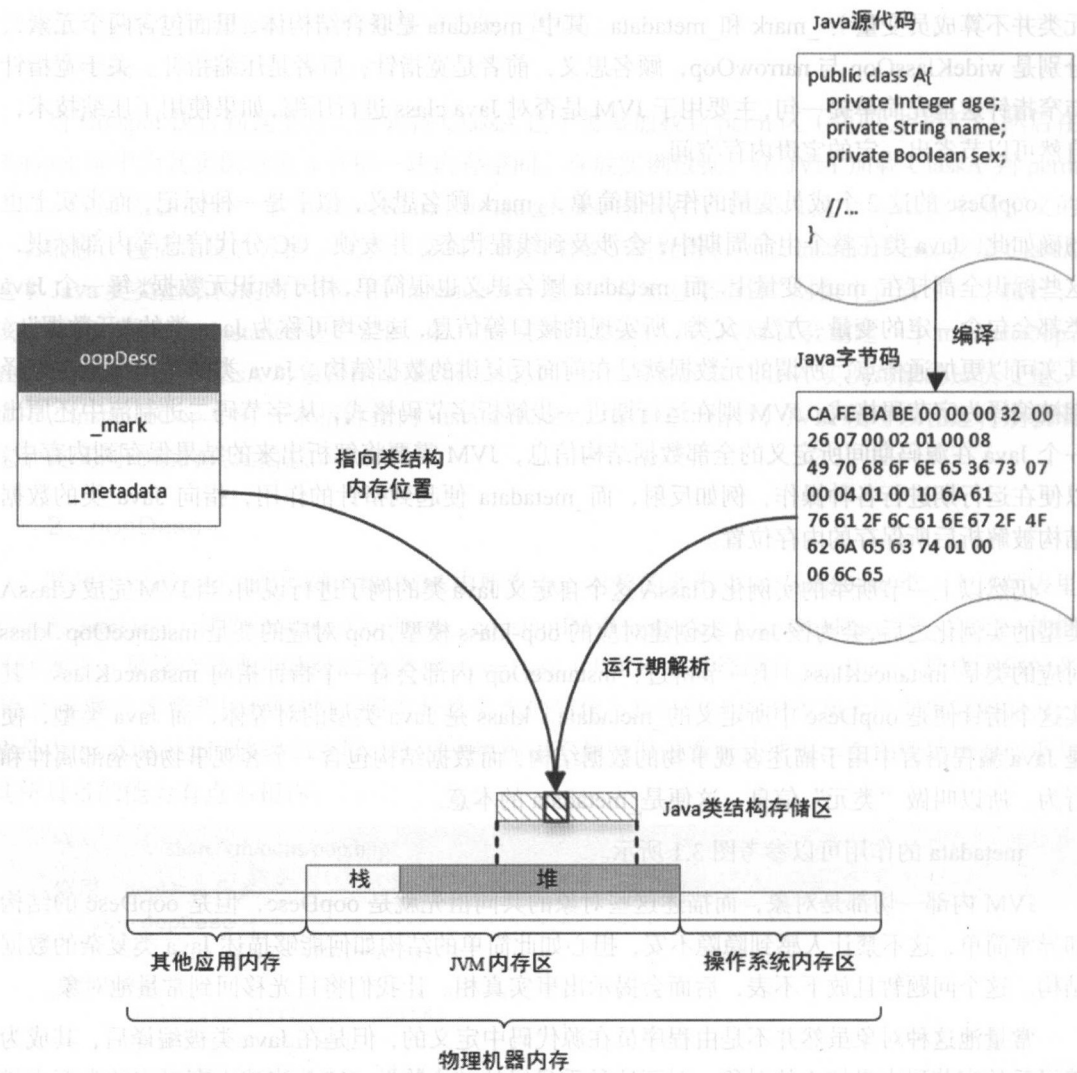


图 3.1 oopDesc 的 _metadata 指向

3. Java 结构与其他编程语言结构之比较

当 C、C++ 和 Delphi 等程序被编译成二进制程序后，原来所定义的高级数据结构都不复存在了，当 Windows/Linux 等操作系统（宿主机）加载这些二进制程序时，是不会加载这些语言中所定义的高级数据结构的，宿主机压根儿就不知道原来定义了哪些数据结构、哪些类，所有的数据结构都被转换为对特定内存段的偏移地址。例如 C 中的 Struct 结构体，被编译后不复存

在，汇编和机器语言中没有与之对应的数据结构的概念，CPU 更不知道何为结构体。C++和 Delphi 中的类概念被编译后也不复存在，所谓的类最终变成内存首地址。而 JVM 虚拟机在加载字节码程序时，会记录字节码中所定义的所有类型的原始信息（元数据），JVM 知道程序中包含了哪些类，以及每个类中所关联的字段、方法、父类等信息。这是 JVM 虚拟机与操作系统最大的区别所在。

正因为 JVM 需要保存字节码中的类元信息，所以 JVM 最终就自然而然地演化出了 OOP-KLASS 这种二分模型，KLASS 用于保存类元信息，而 OOP 用于表示 JVM 所创建的一类实例对象。KLASS 信息被保存在 PERM 永久区，而 OOP 则被分配在 HEAP 堆区。同时 JVM 为了支持反射等技术，必须在 OOP 中保存一个指针，用于指向其所属的类型 KLASS，这样 Java 开发者便能够基于反射技术，在 Java 程序运行期获取 Java 的类型信息，例如反射 Java 类的类型、父类、字段、方法等信息，而这是其他很多语言所不具备的能力。也正因为 JVM 的这种能力，让程序员能够非常方便地开发出具有运行时动态特性的程序，例如可以根据类型来设计更为抽象和优雅的工厂模式，例如可以在运行期动态创建一个新的类型并实例化，同时执行其方法（ASM 字节码编程技术）。由于这些特性，使得 Java 语言成为了互联网时代的各种中间件、各种框架实现的首选语言，进而有力地促进了大数据时代的架构发展。虽然 Java 由于天然的缺陷，在运行性能、内存占用等方面无法与本地语言相媲美，但是在互联网工业时代，生产效率是第一生产力，对于一门编程语言，没有什么能够比易学、易用又能够迅速开发出一款不错的工业产品更具有吸引力。尤其在移动互联网时代，整个人类社会要的是能够在短时间内生产出足够多样化的，能够满足各类日常需求的软件产品，而随着硬件越来越廉价，JVM 在性能和空间两方面的缺点得以弥补。可以说，JVM 的成功不是偶然的，它是整个人类社会发展的必然结果，是 IT 行业自我革命、自我快速发展的必然结果，JVM 是人类对 IT 生产力提升的迫切需求与软件编程门槛高这一对矛盾相互促进而演化出的折中方案。虽然 JVM 不够完美，但已然足够。

3.5 大端与小端

Java 是跨平台的语言，并且支持丰富多样的数据类型。但是在不同的平台上，数据在寄存器、内存、磁盘上的存储格式并不相同——准确地说，是数据存储的顺序不同。这种不同的存储顺序衍生了计算机底层的 2 个概念——大端与小端。

以 JVM 解析 Java class 字节码文件中的魔数为例进行分析。魔数占 4 字节，并且位于字节码文件头部。JVM 解析魔数，只需读取出字节码流开始的 4 字节即可。JVM 读取魔数的代码如下：

清单：/src/share/vm/classfile/classFileParser.cpp

作用：魔数解析

```
u4 magic = cfs->get_u4_fast();
```

cfs 的类型是 ClassFileStream，当 JVM 能够运行到 classFileParser.cpp:: parseClassFile()时，JVM 便已读入当前正在解析的 Java 字节码的文件流。

ClassFileStream 的结构如下：

清单：/src/share/vm/classfile/classFileStream.hpp

作用：classFileStream 类结构

```
class ClassFileStream: public ResourceObj {
private:
    ul*   _buffer_start;
    ul*   _buffer_end;
    ul*   _current;
    char* _source;
    bool  _need_verify;
}
```

这几个字段的含义都相当清晰明了。其中，_current 指针字段值指向 Java 字节码流中当前已经读取到的位置，当 ClassFileStream 类刚刚初始化时，_current 指针指向 Java 字节码流的第 1 个字节所在的内存位置，在 JVM 解析字节码的后续流程中，随着解析的进行，该指针不断向前移动。由于魔数被第一个解析，因此解析之后，_current 指针往前移动 4 字节步长，我们看 cfs->get_u4_fast()的逻辑：

清单：/src/share/vm/classfile/classFileStream.hpp

作用：从字节码流中读取 4 字节内容

```
u4 get_u4_fast() {
    u4 res = Bytes::get_Java_u4(_current);
    _current += 4;
    return res;
}
```

注意，在这个方法中，由于从字节码流中读取了 4 字节的内容，因此 _current 的值被增加了 4，由于 _current 是 ul*类型的指针，该指针所指向的数据类型是一个字节，因此将 _current 的值增加 4，其实就是增加了 4 字节的长度（相信稍微有点 C 语言指针基础的同学都能看得懂）。

在这个方法里，最关键的就是 u4 res = Bytes::get_Java_u4(_current)这句代码，这句代码才真正执行了从字节码流中读取 4 字节的任务，该方法是一个 CPU 架构相关的接口，在 x86 架构上的逻辑如下：

清单: /src/cpu/x86/vm/bytes_x86.hpp

作用: 从内存指定位置读取 4 字节并将其转换为 JVM 内部 u4 类型

```
static inline u4 get_Java_u4(address p) {
    return swap_u4(get_native_u4(p));
}
```

可以看到, 在 `get_Java_u4()` 方法内部连续调用了 2 个函数, 分别是 `get_native_u4()` 和 `swap_u4()`。先看 `get_native_u4()` 方法, 该方法定义如下:

清单: /src/cpu/x86/vm/bytes_x86.hpp

作用: 从内存指定位置读取 4 字节并转换为 u4 类型

```
static inline u4 get_native_u2(address p) {
    return *(u4*)p;
}
```

这个方法内部直接返回 `*(u4*)p`, 按照取值运算符的运算优先级, 这句表达式的运算顺序为 `*((u4*)p)`, 即先将指针 `p` 转换为 `u4*` 类型的指针, 接着对转换后的指针进行取值运算, 同时将运算后的结果转换为 `u4` 类型。

在这里有一个很关键的问题需要注意, 由于 Java 字节码流通常比较大, 随便一个简单的 Java 类都会包含几百字节, 稍微复杂点就需要以 KB 来计了, 而当 JVM 刚开始解析魔数时, 此时 `p` 指针指向字节码流的最开始的内存位置, 后面还有几百字节, 因此使用一个指向首地址的指针可以循环读取后面的几百字节。而每次根据一个指针所能读取到的字节数则取决于指针的类型, 如果指针是 `char*`, 则通过 `*p` 可以读取 1 字节的内容; 如果指针是 `int *`, 则通过 `*p` 可以读取 4 字节的内容。在 JVM 解析魔数时, 原本入参是 `cfs._current` 变量, 其类型是 `u1*`, 但是在 `get_native_u2()` 方法里被强制转换为了 `u4*`, 这相当于将指针的范围扩大, 这样通过 `*p` 能够一次读取 4 字节内容 (这段内容很基础, 专业人士不要嫌弃哟, 毕竟本书是面向广大 Java 道友的, 有关 C/C++ 的基础知识讲解, 笔者觉得对没有 C/C++ 基础的道友而言可能很实惠。同样, 本书在解析其他源码时, 也会经常顺带着介绍一些 C/C++ 的基础知识)。

而事实上, JVM 内部自定义了好几种基本类型, 包括 `u1`、`u2`、`u4`、`u8`, 它们所代表的字节位数分别是 1、2、4、8。每次 JVM 从 Java 字节码流中读取相应位数的字节时, 只需将 `cfs._current` 指针转换为对应的 `u1*`、`u2*`、`u4*`、`u8*` 类型即可。但是早期的 CPU, 并不是随随便便能从指定的内存位置访问任意字节长度的数据, 早期的 CPU 是严格按照字节对齐的要求读取的, 只是从 x86 开始便能支持了。

从 Java 字节码文件中解析出魔数信息很简单, 但是一旦将一个简单的问题放到一个底层的、跨平台的上下文环境中时, 问题便立刻变得复杂。魔数的解析便是一个活生生的例子, 其复杂性全部体现在最关键的 `swap_u4()` 函数中。该函数是跨平台的, 因此需要解决兼容性, 对于 x86

平台，该函数定义如下：

清单：/src/os_cpu/linux_x86/vm/bytes_linux_x86.inline.hpp

作用：大端小端转换

```
inline u4 Bytes::swap_u4(u4 x) {
#ifdef AMD64
    return bswap_32(x);
#else
    u4 ret;
    __asm__ __volatile__ (
        "bswap %0"
        : "=r" (ret)      // output : register 0 => ret
        : "0" (x)         // input  : x => register 0
        : "0"             // clobbered register
    );
    return ret;
#endif // AMD64
}
```

这是一段内嵌了汇编的 C 函数，该函数的作用是进行大端和小端的转换。对于平常以应用开发为主的 C 和 C++ 程序员，以及以使用 Java 语言为主的开发者们，可能并不了解大端和小端的概念，而在网络协议、跨平台兼容性等开发领域，则大端和小端是必须要掌握的一项基本功。下面笔者尝试从简单的概念入手，由浅入深地介绍大端和小端的概念、由来及应用。

3.5.1 大端和小端的概念

关于“大端”和“小端”名词的由来，有一段有趣的故事：

故事来自于 Jonathan Swift 的《格列佛游记》。Lilliput 和 Blefuscu 这两个强国在过去的 36 个月中一直在苦战。战争的原因，大家都知道，吃鸡蛋的时候，原始的方法是打破鸡蛋较大的一端，可那时的皇帝的祖父由于小时候吃鸡蛋，按这种方法把手指弄破了，因此他的父亲，就下令，命令所有的子民吃鸡蛋的时候，必须先打破鸡蛋较小的一端，违令者重罚。然后老百姓对此法令极为反感，期间发生了多次叛乱，其中一个皇帝因此送命，另一个丢了王位，产生叛乱的原因就是另一个国家 Blefuscu 的王国大臣煽动起来的，叛乱平息后，就逃到这个帝国避难。据估计，先后几次有 11 000 余人情愿死也不肯去打破鸡蛋较小的端吃鸡蛋。这个其实在讽刺当时英国和法国之间持续的冲突。Danny Cohen，一位网络协议的开创者，第一次使用这两个术语指代字节顺序，后来就被大家广泛接受。

大端和小端的概念之所以由一位网络协议开创者提出,是因为其实大部分人在实际的开发中都很少会直接和字节序打交道,唯有在跨平台以及网络程序中,才会涉及到一个叫做“字节序”的问题,并且这是一个必须被考虑的基础性问题。字节序,顾名思义,就是指字节的顺序,通俗而言就是数值大于一个字节类型的数据在内存中的存放顺序(一个字节的数当然就无需谈顺序的问题了)。在各种计算机体系结构中,对于字节、字等的存储机制有所不同,通信双方交流的信息单元(比特、字节、字、双字等)的存储顺序不同,因此需要考虑双方数据的传送顺序。如果传送顺序达不成一致,通信双方将无法进行正确的编/译码从而导致通信失败。目前在各种体系的计算机中通常采用的字节存储机制主要有两种:Big-Endian 和 Little-Endian,翻译过来就是所谓的大端和小端。

标准的 Big-Endian 和 Little-Endian 的定义如下:

- ◎ Little-Endian 就是低位字节排放在内存的低地址端,高位字节排放在内存的高地址端。
- ◎ Big-Endian 就是高位字节排放在内存的低地址端,低位字节排放在内存的高地址端。
- ◎ 网络字节序, TCP/IP 协议中使用的字节序通常称为网络字节序, TCP/IP 各层协议将字节序定义为 Big-Endian。

如果之前没有接触过大端和小端,可以看个例子,以建立起对大端和小端概念的初步认识。例如一个值为 0x01020304 的 int 整型数据写入地址为 0x005071 的内存位置,由于该整数占 4 字节,因此需要连续使用 4 个存储单元来存储这个整数,但是存储的方式可以有两种,其中第一种策略是整数的高位字节存储在这 4 个连续存储单元的高位存储单元,整数的低位字节存储在低位存储单元。先解释下什么是整数的高低位(懂的同学可以直接略过),例如 0x01020304,其对应的十进制数是 16909060,对于这个十进制数而言,左边的数字是高位,右边的数字是低位,同样,对于十六进制 0x01020304 而言,也是左侧的代表高位,右侧的代表低位。

按照第一种方式来存储,内存布局如下:

内存地址:	0x005071	0x0050712	0x005073	0x005074
存储的值:	0x04	0x03	0x02	0x01

可以看到,在这种存储策略下,数据增长的方向与内存地址增长的方向是相同的。这种方式不方便人类阅读,但是机器读起来却着实很“爽”,因为这样的存储顺序与机器的解读顺序是一致的。这种存储策略就叫做“小端”。

对应的,第二种存储策略与之相反,数据的高位字节存储在低位存储单元上,数据的低位字节存储在高位存储单元上。还是刚才的数据和内存位置,按照现在这种策略存储后的内存布

局如下：

内存地址：	0x005071	0x0050712	0x005073	0x005074
存储的值：	0x01	0x02	0x03	0x04

现在这种方式，数据字节增长的方向与内存位置增长的方向是相反的，这种方式有点类似于字符串的存储顺序，并且也很符合人类的阅读习惯。这种策略就叫做“大端”。

3.5.2 大小端产生的本质原因

有时候计算机也是挺幽默的，也会玩“时光向左，幸福向右”之类浪漫的事。但是计算机实在不是吃饱了撑的玩这种闲情逸致，内存存储顺序的向左向右，实则是由寄存器引起的。

在计算机体系结构中，内存由存储单元构成，一个存储单元的长度是一个字节，即每个存储单元都对应着一个字节，能够存储 8 比特数据。但是在 C 语言和很多其他高级语言中，除了 8 比特的 char 之外，还有 16 比特的 short 型、32 比特的 int 型与 64 比特的 long 型（int 和 long 具体所占二进制位数要看具体的编译器和 CPU 平台架构）。虽然物理内存的存储单位是 1 字节，但是现代计算机总线线宽和寄存器的宽度往往都大于 1 个字节，对于位数大于 8 位的处理器，例如 16 位或者 32 位的处理器，寄存器宽度都是大于一个字节的，这就造成寄存器宽度与内存存储单元宽度之间的不一致性。在软件程序的很多操作中，都会涉及数据在内存和寄存器之间的传送，例如，如果你的 C 语言程序中包含 `int x=26` 这样的代码，那么最终 CPU 需要先为 x 分配堆栈内存空间，然后将 26 这个立即数传入寄存器，再通过寄存器传送到 x 所在的内存位置。或者如果你的 C 语言程序中包含 `int y=x` 这样的代码，那么 CPU 需要先将 x 的值从内存读取到寄存器，再将数据从寄存器传送到 y 所在的内存位置。在计算机中，并不支持直接将数据在不同的内存之间传送，更不支持将数据直接从内存传送到外部设备，例如磁盘或网络端口。CPU 唯一支持不同部件之间的直接数据传送只有寄存器到寄存器了。由于在高级编程语言（相对于汇编语言而言）中并不能直接操作寄存器，所有的数据传送的指令以及针对寄存器读写的指令都被封装成面向变量的编程，而变量的存储介质是内存，因此可以这么讲，高级编程语言中的所有数据传送指令都必须经过寄存器的中转。

寄存器的宽度越大，就意味着 CPU 传送数据的能力越强，如果寄存器只能容纳 8 比特，那么 CPU 一次指令只能传输 1 字节，而如果寄存器能够容纳双字节，那么一次 CPU 指令就能传输 2 字节，这无形中提升了效率。虽然寄存器的宽度可以提高，但是内存存储单元的宽度却是一直不变的，一直都只有 1 字节，因此对于宽度达到双字节的寄存器可以一次性从内存中读取 2 个连续的存储单元的值，或者一次性向 2 个连续的内存存储单元写入数据。

对于一个占 2 字节的整数,例如 0x0102,数据本身是区分高字节和低字节的,靠近左侧的字节为高字节,反之则为低字节。而一个双字节的寄存器也会区分高低位,对高低位的不同定位会带来数值结果的不同。假设双字节的寄存器中从左至右所存储的字节分别是 0x01 和 0x02,如果将寄存器的左端定位为高字节端,则寄存器所存储的数值就是 0x0102,对应十进制的 258。而如果将寄存器的右端定位为高字节端,那么寄存器所存储的值就是 0x0201,对应的十进制数是 513。由此可见,以相同顺序存储的同样一段数据,如果所标定的高低位不同,则最终所代表的数值是完全不同的。由于不同厂家所生产的 CPU 标准不同,因此大家对于究竟将寄存器的左端还是右端标定为高字节位,并没有一个统一的标准。目前 Intel 的 80x86 系列芯片是唯一还在坚持使用小端的芯片,而 MIPS 和 ARM 等芯片要么全部采用大端的方式储存,要么提供选项支持大端——可以在大小端之间切换。

因此,大小端的问题,本质上是由寄存器引起的(当然,软件系统也能引起所谓的大小端问题,完全看人为的定义),并最终在内存的存储顺序上得以反映出来。举个例子来说明。

假设在 C 程序中包含 `int x=0x0102` 这样一句代码,在大端 CPU 架构平台上运行到这句代码时,CPU 首先将立即数 0x0102 保存到双字节的寄存器中(假设寄存器宽度为 2 字节),由于以大端的方式存储,因此寄存器中从低位到高位所存储的字节分别是 0x01 和 0x02。接着 CPU 将寄存器中的数据传送到 `x` 变量所在的内存位置,传送后,`x` 变量所在的 2 个连续的内存存储单元中的值从低位到高位也分别是 0x01 和 0x02。

而对于同样的程序,如果运行在小端 CPU 架构平台上,则寄存器从低位到高位所存储的字节分别是 0x02 和 0x01,将其拼接起来得到的数据是 0x0201,与原来的数据 0x0102 的字节位正好相反,于是最终保存到内存中,在内存中从低位到高位所保存的数值就是 0x02 和 0x01。

3.5.3 大小端验证

x86 架构的 CPU 是属于小端 CPU,大部分 Windows 用户都使用这种架构的 CPU,笔者的机器也是这种。可以写程序来验证,示例程序如下,使用 C 语言编写:

清单: `/src/os_cpu/linux_x86/vm/bytes_linux_x86.inline.hpp`

作用: 大端小端转换

```
#include <stdio.h>
int main()
{
    short x = 1;
    char c = *(char*)&x;
    if(c == 1)
    {
```

```

    printf("litte endian\n");
}
else
{
    printf("big endian\n");
}
return 0;
}

```

这段代码测试的原理很简单，由于变量 `x` 的数据类型是 `short`，在 32 位平台上占 2 字节，其值转换为十六进制是 `0x0001`，如果当前 CPU 是大端类型，则最终保存到内存后，内存首地址的那个存储单元中所存储的值一定是 `0x00`，这是因为大端 CPU 的特点就是数据的高位字节存储在低位内存单元。反之，如果 `x` 的内存首地址所存储的数值是 `0x1`，则代表当前 CPU 是小端。而要拿到变量 `x` 的内存首地址所在的存储单元中所存储的值，只需像本示例中那样，通过 `char c = *(char*)&x` 来获取。这句代码的含义是，首先通过 `&x` 获取变量 `x` 的内存首地址，获取的结果是一个指针类型，只是指针的类型是 `short*`，这个指针指向的内存范围包含 2 个存储单元，因此需要将其转换为 `char*` 这种指针类型，这样最终通过 `*(char*)` 得到的结果数据类型才是 `char`，否则就变成了 `short` 类型。

如果上面这段测试程序还不够直观，则下面的这段程序一定能够说明问题了：

```

#include<stdio.h>

int main()
{
    char cc[6];
    cc[0]=0x11;
    cc[1]=0x22;
    cc[2]=0x33;
    cc[3]=0x44;
    cc[4]=0x55;
    cc[5]=0x66;

    char *cp=&cc;
    short cs=(short*)cp;
    printf("cs=%d\n", cs);

    return 0;
}

```

在本示例中，通过数组 `cc` 往内存中连续写入 6 字节的内容，从内存地址的低位到高位分别是 `0x11`、`0x22`、`0x33`、`0x44`、`0x55`、`0x66`。通过 `char *cp=&cc` 拿到 `cc` 数组的内存首地址，这个首地址所存放的数值是 `0x11`。接着通过 `short cs=(short*)cp` 获取从 `cc` 内存首地址开始的连续

2 个存储单元的值，并将其转换为 short 类型。如果是在大端 CPU 平台上，则 short 的值应该是 0x1122，对应的十进制数值是 4386，但是很可惜并不是这个数值，而是 8721，8721 所对应的十六进制数正是 0x2211。这说明当前 CPU 是小端，而事实上也是的，因为是 x86 平台。之所以小端 CPU 会将 0x1122 这样的内存数据解读成 0x2211，就是因为小端 CPU 认为内存中的低位代表数据的高位字节，而内存中的高位则代表数据的低位字节，因此最终 CPU 会认为这段内存中的数值是 0x2211。

3.5.4 大端和小端产生的场景

虽然大端小端的问题在内存、寄存器、计算机总线甚至软件中到处都存在，但是得益于整个软硬件架构的良好设计，因此在日常编程开发中，大多数程序员都不需要去关注这个问题。在单机上，不管是往内存中写入数据还是从内存中读取数据，由于所采用的标准都是同一套，要么全部是大端模式，要么全部是小端模式，因此不会产生大小端数据转换的问题。例如对于下面这段程序：

```
int x = 0x0102;
int y = x;
```

程序运行时，先将 0x0102 赋值给变量 x，再将 x 的内存值赋值给 y。虽然在这期间，数据会发生多次从内存到寄存器，再从寄存器到内存的传送，但是并不会产生混乱，只要所使用的大端和小端模式相同。如果使用的是小端模式，则最终变量 x 所占用的 4 个连续的内存存储单元从低位到高位所存储的数据分别是 0x01、0x00、0x00、0x00，寄存器在传送数据的过程中暂存数据时，寄存器从低位到高位所存储的数据也是 0x01、0x00、0x00、0x00，而最终寄存器往变量 y 所在内存写入的数据在内存中的存储顺序也是 0x01、0x00、0x00、0x00。当你想打印变量 x 或变量 y 的值时，虽然存储顺序与实际数据的高低位完全相反，但是 CPU 的逻辑运算器是清楚这种格式的，并会将其字节反转后拼装出正确的结果显示出来。

因此在单机上由同一种模式进行数据读写时，并不会产生对数据识别的不同，这也是为什么绝大部分程序员在日常开发中都不会接触到大端和小端问题的原因。

但是当关注网络传输和文件共享时，由于数据在网络的一端写入，而由网络的另一端读取，网络两端的 CPU 架构并不总是相同的，很可能一端使用了大端模式，而另一端使用了小端模式。在这种场景下，如果不进行大端与小端模式的转换，则数据必定会出现不一致。例如下面这段示例程序，往文件中写入 int 类型的数据：

```
#include <stdio.h>
#include <assert.h>
```

```

void main()
{
    short test;
    FILE* fp;

    // (0x41 的 ASCII 码对应字符 A, 0x42 的 ASCII 码对应字符 B)
    test = 0x4142;
    if ((fp = fopen ("/home/test.log", "wb")) == NULL)
    {
        assert(0);
    }

    fwrite(&test, sizeof(short), 1, fp);
    fclose(fp);
}

```

在本段代码中，往文件中写入了一个 `short` 类型的数据，由于一个 `short` 类型的数据占用 2 字节的内存空间，因此最终往文件里写入了 2 字节。而这 2 字节所对应的 ASCII 字符分别是 A 和 B，因此写入文件后，使用记事本或者在 Linux 上使用 `vim` 或 `gedit` 打开文件时，应该看到的是字符 A 和 B，而不是数值（这些文本处理器默认使用 ASCII 字符集编码）。

由于笔者的机器是 x86 架构，而 x86 属于小端模式，因此当 CPU 执行 `short test=0x4142` 这行代码时，最终变量 `test` 在内存中从低位到高位存储内容便会与 0x4142 的字节高低位顺序相反，变成 0x4241。这导致最终写入到文件中的数据顺序也是 0x4241，写入后在 x86 平台上使用文本编辑器打开时，会发现显示的内容是 BA，而不是 AB。如果在大端 CPU 架构上使用文本编辑器查看本文件，则结果仍为 BA，这是因为文本编辑器是按字节逐个读取内容的，而寄存器在读取一个字节时，不会发生字节之间的乱序。

现在再写一段程序来读取刚才写入的文件，示例程序如下：

```

#include <stdio.h>
#include <assert.h>

#define MAXLEN 1024

int main()
{
    FILE *fp;
    fp = fopen("/home/test.log", "rb");

    unsigned char buf[MAXLEN];

    if( fp == NULL)
    {
        printf("%s, %s", "错误", "not exit\n");
        return 0;
    }
}

```

```

}

int rc;

while( (rc = fread(buf,sizeof(unsigned char), MAXLEN,fp)) != 0 )
{
    printf("%s\n","read start");
    int readInt = *(int*)&buf;
    printf("readInt = %d\n", readInt);
}
printf("%s\n","read over");

fclose(fp);

return 0;
}

```

这段程序从刚才所写入的文件中读出 1024 个字节到缓冲区 buf 中，并强制将指针&buf 转换成 int*类型，最终再通过 int*指针获取 int 值。

同样在 x86 这种小端模式的 CPU 架构平台上运行这段程序，结果打印出十进制数据 16706，其对应的十六进制正好是 4142，这与原本往文件里写入的数据是完全一致的。

在本例中，由于写入和读取文件的程序都运行在小端 CPU 架构平台上，因此虽然写入文件后的数据高低字节位置被反转了，但是在同样的 CPU 架构平台上能够识别出正确的数值。但是，如果将读取文件的这段示例程序放在大端 CPU 架构平台上运行，则会发现这段程序最终会打印出 16961，即对应十六进制 0x4241。这与将上个示例程序在小端 CPU 平台上运行后写入文件并在大端 CPU 架构平台上使用文本编辑器打开后显示的字符依然是 BA 不同，这是因为文本编辑器使用 ASCII 字符集进行编码，文本编辑器逐字节读取磁盘文件里的内容，不会将文件里的字符合并转换成一个 int 类型的数据，因此不会产生字节序反转，而本示例需要 CPU 识别多个字符合并后所代表的某种数据类型的数值，在这个过程中，由于大端 CPU 与小端 CPU 所标定的高低位相反，因此对于由 2 字节所合并出来的同一个数据的识别也必定不同。

所以在编写网络协议或者编写跨平台的程序时，大端和小端的问题就显得尤其突出，开发者必须要清楚所谓大小端问题产生的根本原因并采取适当的措施进行解决，否则必定产生乱序，导致数据不一致。

但是，并不是所有的网络通信场景都需要考虑大小端问题，例如下面的示例：

```

#include <stdio.h>
#include <assert.h>

void main( )
{

```



```

// 注意：这里将 short 类型换成了 char[] 数组
char test[2];
FILE* fp;

// 0x41 的 ASCII 码对应字符 A, 0x42 的 ASCII 码对应字符 B
test[0] = 0x41;
test[1] = 0x42;
if ((fp = fopen ("/home/test.log", "wb")) == NULL)
    assert(0);
fwrite(&test, sizeof(short), 1, fp);
fclose(fp);
}

```

在本示例中，写入文件的不再是 `int` 整数，而是变成了 `char` 数组，其实就是逐字节写入。依然在 x86 平台上运行本程序，运行后使用文本编辑器打开写入的文件，会发现编辑器里显示的内容变成“AB”，而不再是“BA”。为什么这一次 CPU 没有对数据高低字节位进行反转呢？其实这就涉及一个核心的问题：究竟什么场景下才会产生大小端模式带来的字节序反转问题？

其实这一问题的答案在前文已经说得很清楚。要回答一个问题，首先就要明白问题产生的本质原因。大小端问题产生的根本原因是，当寄存器需要读取或写入超过一个字节长度的数据时，由于不同 CPU 所认定的寄存器的高低位不同（只有 2 种认定，并且认定结果完全相反），而产生了所谓大小端模式。只要不使问题产生的条件成立，那么问题自然不会产生。由于程序在处理 `char` 类型的数据时，寄存器只需要读写 1 字节宽度的数据，因此自然不会出现所谓的字节序反转问题。这就是本示例能够不受大小端模式影响而始终生成同样顺序的字节的原因。而在上一个示例中，我们往文件里写入的是 `int*` 类型的数据，但是字节序反转并不是发生在 CPU 将变量写入文件的阶段，而是发生在 CPU 为变量写入数值的前一阶段，即 CPU 将立即数读进寄存器的阶段。因为是小端 CPU，因此寄存器读取到超过 1 个字节宽度的 `int` 类型数据时便发生了字节序反转。由于在寄存器中数据的字节序被反转，因此最终写入变量内存时的字节序也是反转的，由此导致最终写入文件的字节序依然是反转的。

其实，在这个过程中，有一个重要的点值得关注，那就是无论在大端还是小端，调用系统 API `fwrite()` 和 `fread()` 进行读写时，似乎并不受大小端模式的影响。理解了上面的道理之后，对于这一现象就能解释了，唯一的原因就是 `fwrite` 与 `fread` 在底层也是逐字节读取和写入的，因此并不会产生字节序反转。

3.5.5 如何解决字节序反转

前面讨论了大小端的概念、产生原因及现象，那么，如果真的面临大小端的现实场景，该

如何解决则是一个重中之重的问題。例如，假设有一天你需要使用 C 语言开发一个网络通信协议，该协议要求兼容主流平台，那就必须要处理大小端问题。

在 Linux 平台，可以调用 `bswap` 这个指令进行字节序反转。我们在前文举了一个例子，定义一个包含 6 个元素的 `char` 数组，然后将其前两个字节合并转换成一个 `short` 类型的数据，现在我们改造这个例子，同时支持字节序反转：

```
#include<stdio.h>

int swap_u4(int x) ;

int main(){
    char cc[6];
    cc[0]=0x11;
    cc[1]=0x22;
    cc[2]=0x33;
    cc[3]=0x44;
    cc[4]=0x55;
    cc[5]=0x66;

    char *cp=&cc;
    short cs=(short*)cp;
    printf("cs=%d\n", cs);

    int lls=(int*)cp;
    printf("lls=%d\n", lls);

    int lld=swap_u4(lls);//这里对字节序进行反转
    printf("lld=%d\n", lld);

    int xx=0x01020304;
    char iic=(char*)&xx;
    printf("iic = %d\n", iic);

    return 0;
}

int swap_u4(int x) {
#ifdef AMD64
    return bswap_32(x);
#else
    int ret;
    __asm__ __volatile__ (
        "bswap %0"
        : "=r" (ret) // output : register 0 => ret
        : "0" (x) // input : x => register 0
```

```
        : "0" // clobbered register
    );
    return ret;
#endif // AMD64
}
```

在本例中，定义了一个函数 `swap_u4()` 用于反转字节序。在小端 CPU x86 架构平台上运行本程序，打印结果如下：

```
lls=1144201745
lld=287454020
```

`lls` 对应的十六进制是 `0x44332211`，而 `lld` 对应的十进制则是 `0x111223344`。由此可见，`lld` 的值被正确地还原了出来，与原始的输入值完全相等。而 `lld` 之所以能够被完整还原出来，是因为调用了字节序反转函数 `swap_u4()`。

3.5.6 大小端问题的避免

大小端问题的本质是由于计算机底层硬件的问题，因此显得比较复杂，但是在宏观表象上，却比较容易解释和理解，绝大多数书籍和网络博客阐述的重点往往也是表面原因和现象。

但是只要遵循下面两种方式处理数据、文件、网络传输，便可以无视大小端模式：

- ◎ 在单机上使用同一种编程语言读写变量、读写文件、进行网络通信，所读到的字节序与所写入的字节序相同，反之亦成立。
- ◎ 在分布式场景下，使用同一种编程语言，在同样大小端模式的不同机器上所读写的文件与网络信息，字节序相同。

例如，在网络环境中，在一台机器上写入文件，在另一台机器上读取文件，如果这两台机器都是大端模式或者都是小端模式，则只要读取和写入时均使用同样的编程语言便能保持所读与所写的字节序是相同的。这一点对于理解 JVM 的字节序处理很重要。

3.5.7 JVM 对字节码文件的大小端处理

在讲述关于 Java 字节码文件的大小端问题处理之前，还有一个问题需要特别说明，那就是大小端问题不仅存在于计算机硬件体系中，软件中也同样存在。当然，软件中的大小端问题多是被编译器处理了，所以在绝大多数情况下，软件开发者并不需要特别关注大小端问题。Java 编译器同样在后端默默地处理了这个问题，Java 所输出的字节信息全部是大端模式，这一点对于理解 JVM 在解析字节码信息时的处理策略至关重要。

JVM 在解析魔数及 Java 类的其他结构信息时,均需要读取字节码信息。Java 源代码一般由编译器被编译为字节码文件,而 Java 编译器本身一般也都是用 Java 语言编写而成的,因此 Java 编译器在对 Java 源程序进行语法树分析并将分析结果写入字节码文件时,字节码文件中的信息存储便是大端模式,例如对于魔数,字节码文件的写入顺序一定是如下这样:

```
0xCA 0xFE 0xBA 0xBE
```

这种写入文件的顺序不会受计算机硬件体系到底是 大端模式 还是 小端模式 的影响。这是 Java 与其他编程语言的一个重要区别,例如,如果由用 C 语言开发的编译器来编译 Java 源代码,那么这种编译器在小端机器上生成的字节码文件中的魔数的写入顺序基本会变成下面这样:

```
0xBE 0xBA 0xFE 0xCA
```

这是因为,魔数在 C 语言中可以使用一个 int 类型的变量表示, C 语言在将这个 int 型变量写入字节码文件之前,首先需要将魔数信息写入该变量,而由于硬件体系属于小端模式,因此从寄存器写入内存时,魔数的字节序便已经发生反转。

既然 Java 编译器所生成的字节码文件并不会因为计算机架构的大小端模式而受到影响,那么 JVM 从字节码文件中解析魔数时,为何还要处理字节序呢?在上文讲到,在分布式环境下,要避免大小端问题,只需使网络中的各个计算机节点的大小端模式都保持一致,并且所使用的编程语言也保持一致即可。而 Java 字节码的读写场景对这两个条件都不符合,首先是读写问题,Java 编译器一般是由 Java 开发的,因此字节码文件的写入可以认为是由 Java 语言完成的,而读取字节码文件的是 JVM, JVM 是由 C 与 C++ 混合写成的,因此 Java 字节码文件的写入端与读取端属于两种不同的编程语言。接着看计算机节点的大小端模式的一致性问题。由于 Java 语言的跨平台性,因此对于一段已经编写好的 Java 源代码,既可以在 Windows 上编译打包,也可以在 Linux 或者其他平台上编译打包。同样,读取 Java 字节码的 JVM 可能运行于 Windows 平台,也可能运行于 Linux 平台,因此 Java 字节码文件可能在 Windows 平台上写入,而在 Linux 平台上被读取。但是 Java 字节码文件的字节序并不受大小端模式的影响,这是由于 Java 语言本身属于大端模式,因此 Java 语言所写入的文件的字节序全部按照大端模式进行存储。如此看来,可能引起 Java 字节码文件的字节序读取不一致的是读取端的编程语言的大小端模式。JVM 由 C 和 C++ 编写,而 C 和 C++ 的大小端模式默认情况下与计算机硬件平台的大小端模式保持一致,因此最终又完全取决于读取端所在的计算机硬件平台的大小端模式。如果读机器属于大端模式,则最终所读取到内存中的魔数信息便是 0xCAFEBAE;反之,如果读机器属于小端模式,则最终所读取到的魔数信息便是 0xBEBAFECA。很显然,如果是后者,则 JVM 将会校验失败,因此如果 JVM 运行于小端机器上,则必须对所读取出来的魔数字节序进行反转。这便是 JVM 最终在 bytes_linux_x86.inline.hpp 中引入 inline u4 Bytes::swap_u4(u4 x)这类函数接口的原因。

事实上，JVM 不仅在解析魔数时需要实现大小端的正确反转，而且在后续解析所有其他信息，诸如版本号、常量池、字段等时，也都实现了大小端的兼容处理。魔数占用 4 字节，因此 JVM 定义了 `swap_u4()` 这样的接口，而除魔数以外的其他字节码信息，有的占用 2 字节，有的占用 8 字节，当然，也有的仅占用 1 字节。对于 2 字节和 8 字节的读取，JVM 同样定义了相应的转换接口，如下：

清单：/src/share/vm/classfile/classFileStream.hpp

作用：大端小端转换 u2 版本

```
inline u2 Bytes::swap_u2(u2 x) {
#ifdef AMD64
    return bswap_16(x);
#else
    u2 ret;
    __asm__ __volatile__ (
        "movw %0, %%ax;"
        "xchg %%al, %%ah;"
        "movw %%ax, %0"
        : "=r" (ret)      // output : register 0 => ret
        : "0" (x)         // input : x => register 0
        : "ax", "0"       // clobbered registers
    );
    return ret;
#endif // AMD64
}
```

清单：/src/share/vm/classfile/classFileStream.hpp

作用：大端小端转换 u8 版本

```
#ifdef AMD64
inline u8 Bytes::swap_u8(u8 x) {
#ifdef SPARC_WORKS
    // workaround for SunStudio12 CR6615391
    __asm__ __volatile__ (
        "bswapq %0"
        : "=r" (x)      // output : register 0 => x
        : "0" (x)       // input : x => register 0
        : "0"           // clobbered register
    );
    return x;
#else
    return bswap_64(x);
#endif
}
#else
// Helper function for swap_u8
```

```

inline u8 Bytes::swap_u8_base(u4 x, u4 y) {
    return (((u8)swap_u4(x))<<32) | swap_u4(y);
}

inline u8 Bytes::swap_u8(u8 x) {
    return swap_u8_base(*(u4*)&x, *(((u4*)&x)+1));
}

#endif // !AMD64

```

可以看到, JVM 并没有定义一个类似于 `swap_u1()` 这样的转换接口, 这是因为在前文分析过, 大小端问题的产生条件是读取或写入超过 1 字节长度的数据, 而如果从文件中一字节一字节地读取, 1 字节码的读取并不会产生乱序行为, 因此 JVM 并不需要针对 `u1` 类型的数据的读取进行大小端的兼容性处理。

本节讲述了大端与小端的概念、产生机制以及 JVM 内部的解决之道。通过这一点更加可以看出, 如果没有 Java 语言, 大家要实现将一个程序部署到不同的硬件平台上, 是一件多么辛苦的事情, 仅仅是大端与小端, 就要花费很多精力去处理。

3.6 本章总结

到了现在, 可以回答本章开始处“摘要”中的问题: 数据结构是什么, 为何要数据结构?

从程序的角度看, 数据结构是若干数据的有机结合, 一个数组、一个链表、一个堆/栈都是数据集, 这些数据在内存位置上有着紧密的联系, 例如, 对于数组而言, 相邻的两个元素在内存位置上也是彼此相邻的; 而对于链表, 内存空间上未必彼此相连, 但是相邻的两个元素中必定有一个元素保存着一个指针指向另一个元素的内存位置。而从人类的角度看, 一个特定的数据结构可以更好地描述客观世界, 例如通过一个 Java 类可以描述一只猫、一个手机, 而通过类的组合则可以描述几乎任意复杂的事物。这便是数据结构的意义所在。

另一个问题是: Java 数据结构的实现机制是什么?

总体而言, Java 的数据结构的实现机制是, 编译时变成字节码, 运行期实现。

Java 为何编译期不支持数据结构? 有两方面的因素: 一是因为 Java 为了实现算法的跨平台性而选择了字节码, 数据结构的实现也必须跨平台, 而不能直接被编译为机器指令; 二是因为 Java 要实现 RTTI, 即运行期类型识别。

换言之，在编译后所生成的字节码文件中，Java 类的数据结构信息其实是被抹掉的，谁也无法一眼就能从二进制格式的字节码文件中看出一个 Java 类的明确结构。但是，字节码文件通过其本身的格式化规范，确保 JVM 能够据此还原出原始的 Java 类的结构。这便是 Java 数据结构的实现机制。

本章详细分析了 JVM 如此实现数据结构的技术必然性。

源代码本 0.0

第 4 章

Java 字节码实战

本章摘要

- ◎ Java 字节码的二进制格式
- ◎ Java 字节码的魔数与版本
- ◎ Java 字节码的常量池
- ◎ Java 字节码的类继承
- ◎ Java 字节码的字段存储
- ◎ Java 字节码的方法格式

我们在前面章节与各位道友一起分享了 JVM 内部调用 Java 方法的核心技术原理，接着从宏观层面谈了 Java 在面向对象、数据结构上所做的技术选择。要想深刻理解 JVM 执行引擎的机制，就必须对 JVM 内部的数据结构有深入了解，而要了解 JVM 内部的数据结构，就必须要了解从 Java 源程序过渡到 JVM 内部数据结构的中间桥梁——Java 字节码。下面就与各位道友一起探讨 Java 字节码文件的格式组成。

4.1 字节码格式初探

相信绝大多数人初次接触字节码相关的知识时，都会觉得比较抽象。因此为了让大家能够真正熟悉并理解 Java 字节码文件的格式，下面将通过一个实际的 Java 类的字节码文件格式分析来进行讲解。如果你对这一部分内容已经非常熟悉，可以跳过本节，这完全不会影响对后续内容的阅读和理解。如果你对这一部分内容仅仅停留在“知道”的层面，那么阅读本节可以让你真正理解字节码。

4.1.1 准备测试用例

所构造的 Java 测试类如下：

清单：Test.java

作用：java 测试类

```
public class Test {
    public int a = 3;
    static Integer si = 6;
    String s = "Hello World!";

    public static void main(String[] args){
        Test test = new Test();
        test.a = 8;
        si = 9;
    }

    private void test(){
        this.a = a;
    }
}
```

这个测试类虽然简单，然而五脏俱全，包含类变量、成员变量、字符串和类成员方法，同时包含一个入口主函数 `main()`，并且在 `main()` 函数中实例化了 `Test` 类。为了简单起见，该类没有继承任何类，没有使用任何类库，这样有助于在分析字节码文件格式时降低难度。

4.1.2 使用 javap 命令分析字节码文件

在 `%JAVA_HOME%\bin` 目录下包含一个 `javap` 命令工具，`javap` 命令能够分析出一个给定的 `java` 类中的字节码信息。这里使用 `javap` 命令来分析上述用于测试的 `Test` 类。首先编译 `Test.java` 类，得到 `Test.class` 文件，进入 `Test.class` 文件所在目录，输入如下命令：

```
javap -verbose Test
```

执行 `javap` 命令后，将输出如下信息：

清单：Test.class

作用：使用 `javap` 命令分析 `class` 字节码文件内容

```
D:\work\projects\person\bin>javap -verbose Test
Compiled from "Test.java"
public class Test extends java.lang.Object
  SourceFile: "Test.java"
  minor version: 0
```

major version: 50

Constant pool:

```

const #1 = class      #2;      // Test
const #2 = Asciz      Test;
const #3 = class      #4;      // java/lang/Object
const #4 = Asciz      java/lang/Object;
const #5 = Asciz      a;
const #6 = Asciz      I;
const #7 = Asciz      si;
const #8 = Asciz      Ljava/lang/Integer;;
const #9 = Asciz      s;
const #10 = Asciz     Ljava/lang/String;;
const #11 = Asciz     <clinit>;
const #12 = Asciz     ()V;
const #13 = Asciz     Code;
const #14 = Method    #15.#17;  //
java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
const #15 = class    #16;      // java/lang/Integer
const #16 = Asciz    java/lang/Integer;
const #17 = NameAndType #18:#19; // valueOf:(I)Ljava/lang/Integer;
const #18 = Asciz    valueOf;
const #19 = Asciz    (I)Ljava/lang/Integer;;
const #20 = Field    #1.#21; // Test.si:Ljava/lang/Integer;
const #21 = NameAndType #7:#8; // si:Ljava/lang/Integer;
const #22 = Asciz    LineNumberTable;
const #23 = Asciz    LocalVariableTable;
const #24 = Asciz    <init>;
const #25 = Method    #3.#26; // java/lang/Object."<init>":()V
const #26 = NameAndType #24:#12; // "<init>":()V
const #27 = Field    #1.#28; // Test.a:I
const #28 = NameAndType #5:#6; // a:I
const #29 = String    #30;      // Hello World!
const #30 = Asciz    Hello World!;
const #31 = Field    #1.#32; // Test.s:Ljava/lang/String;
const #32 = NameAndType #9:#10; // s:Ljava/lang/String;
const #33 = Asciz    this;
const #34 = Asciz    LTest;;
const #35 = Asciz    main;
const #36 = Asciz    ([Ljava/lang/String;)V;
const #37 = Method    #1.#26; // Test."<init>":()V
const #38 = Asciz    args;
const #39 = Asciz    [Ljava/lang/String;;
const #40 = Asciz    test;
const #41 = Asciz    SourceFile;
const #42 = Asciz    Test.java;

```

```
public int a;
static java.lang.Integer si;
java.lang.String s;

static {};
Code:
    Stack=1, Locals=0, Args_size=0
    0:  bipush 6
    2:  invokestatic  #14; //Method
java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
    5:  putstatic  #20; //Field si:Ljava/lang/Integer;
    8:  return

public Test();
Code:
    Stack=2, Locals=1, Args_size=1
    0:  aload_0
    1:  invokespecial  #25; //Method java/lang/Object."<init>":()V
    4:  aload_0
    5:  iconst_3
    6:  putfield  #27; //Field a:I
    9:  aload_0
   10:  ldc  #29; //String Hello World!
   12:  putfield  #31; //Field s:Ljava/lang/String;
   15:  return

public static void main(java.lang.String[]);
Code:
    Stack=2, Locals=2, Args_size=1
    0:  new  #1; //class Test
    3:  dup
    4:  invokespecial  #37; //Method "<init>":()V
    7:  astore_1
    8:  aload_1
    9:  bipush 8
   11:  putfield  #27; //Field a:I
   14:  bipush 9
   16:  invokestatic  #14; //Method
java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
   19:  putstatic  #20; //Field si:Ljava/lang/Integer;
   22:  return
}
```

使用 `javap -verbose` 命令分析一个字节码文件时，将会分析字节码文件的魔数、版本号、常量池、类信息、类的构造函数、类中所包含的方法信息以及类（成员）变量信息。需要注意的是，每一次执行 `javap` 命令所输出的信息内容一定是相同的，但是信息的先后顺序则不保证完

全一致，例如，常量池中的元素编号每次都不保证相同。

javap -verbose 会列出 Java 类中的全部常量池，其格式如下：

清单：javap -verbose 显示常量池的格式

```
Constant pool:
  const #1 = class           #2;      // Test
  const #2 = Asciz           Test;
  const #3 = class           #4;
  .....
```

每一个 const 后面的#号后的数字代表了常量池项在常量池中的索引，当 JVM 在解析类的常量池信息时，常量池项的索引与此一致。

4.1.3 查看字节码二进制

下面使用十六进制工具打开 Test.class 文件，让大家预先感受一下字节码文件的内容到底是什么。打开文件后得到的十六进制文件内容如下（系笔者全手工录入——逐个字符录入）：

清单：project/src/Test.java

作用：Java 测试类，用于跟踪调试 JVM 的代码运行机制

```
00000000 CA FE BA BE 00 00 00 32 00 33 07 00 02 01 00 04 .....2.3.....
00000010 54 65 73 74 07 00 04 01 00 10 6A 61 76 61 2F 6C Test.....java/l
00000020 61 6E 67 2F 4F 62 6A 65 63 74 01 00 01 61 01 00 ang/Object...a..
00000030 01 49 01 00 02 73 69 01 00 13 4C 6A 61 76 61 2F .I...si...Ljava/
00000040 6C 61 6E 67 2F 49 6E 74 65 67 65 72 3B 01 00 01 lang/Integer;...
00000050 73 01 00 12 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 s...Ljava/lang/S
00000060 74 72 69 6E 67 3B 01 00 08 3C 63 6C 69 6E 69 74 tring;...<clinit
00000070 3E 01 00 03 28 29 56 01 00 04 43 6F 64 65 0A 00 >...()V...Code..
00000080 0F 00 11 07 00 10 01 00 11 6A 61 76 61 2F 6C 61 .....java/la
00000090 6E 67 2F 49 6E 74 65 67 65 72 0C 00 12 00 13 01 ng/Integer.....
000000a0 00 07 76 61 6C 75 65 4F 66 01 00 16 28 49 29 4C ..valueOf...(I)L
000000b0 6A 61 76 61 2F 6C 61 6E 67 2F 49 6E 74 65 67 65 java/lang/Intege
000000c0 72 3B 09 00 01 00 15 0C 00 07 00 08 01 00 0F 4C r;.....L
000000d0 69 6E 65 4E 75 6D 62 65 72 54 61 62 6C 65 01 00 ineNumberTable..
000000e0 12 4C 6F 63 61 6C 56 61 72 69 61 62 6C 65 54 61 .LocalVariableTa
000000f0 62 6C 65 01 00 06 3C 69 6E 69 74 3E 0A 00 03 00 ble...<init>....
00000100 1A 0C 00 18 00 0C 09 00 01 00 1C 0C 00 05 00 06 .....
00000110 08 00 1E 01 00 0C 48 65 6C 6C 6F 20 57 6F 72 6C .....Hello Worl
00000120 64 21 09 00 01 00 20 0C 00 09 00 0A 01 00 04 74 d!.....t
00000130 68 69 73 01 00 06 4C 54 65 73 74 3B 01 00 04 6D his...LTest;...m
00000140 61 69 6E 01 00 16 28 5B 4C 6A 61 76 61 2F 6C 61 ain...([Ljava/la
00000150 6E 67 2F 53 74 72 69 6E 67 3B 29 56 0A 00 01 00 ng/String;)V....
00000160 1A 0A 00 01 00 27 0C 00 28 00 29 01 00 04 73 65 ...../...se
```



```

00000170 74 41 01 00 16 28 4C 6A 61 76 61 2F 6C 61 6E 67 tA...(Ljava/lang
00000180 2F 49 6E 74 65 67 65 72 3B 29 56 01 00 04 61 72 /Integer;)V...ar
00000190 67 73 01 00 13 5B 4C 6A 61 76 61 2F 6C 61 6E 67 gs...[Ljava/lang
000001a0 2F 53 74 72 69 6E 67 3B 01 00 04 74 65 73 74 0A /String;...test.
000001b0 00 0F 00 2E 0C 00 2F 00 30 01 00 08 69 6E 74 56 ...../.0...intV
000001c0 61 6C 75 65 01 00 03 28 29 49 01 00 0A 53 6F 75 alue...(I)...Sou
000001d0 72 63 65 46 69 6C 65 01 00 09 54 65 73 74 2E 6A rceFile...Test.j
000001e0 61 76 61 00 21 00 01 00 03 00 00 00 03 00 01 00 ava.! .....
000001f0 05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00 .....
00000200 09 00 0A 00 00 00 04 00 08 00 0B 00 0C 00 01 00 .....
00000210 0D 00 00 00 29 00 01 00 00 00 00 00 09 10 06 B8 .....) .....
00000220 00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 06 .....
00000230 00 01 00 00 00 04 00 17 00 00 00 02 00 00 00 01 .....
00000240 00 18 00 0C 00 01 00 0D 00 00 00 46 00 02 00 01 .....F...
00000250 00 00 00 10 2A B7 00 19 2A 06 B5 00 1B 2A 12 1D .....*...*...*...
00000260 B5 00 1F B1 00 00 00 02 00 16 00 00 00 12 00 04 .....
00000270 00 00 00 01 00 04 00 02 00 09 00 06 00 0F 00 01 .....
00000280 00 17 00 00 00 0C 00 01 00 00 00 10 00 21 00 22 .....!..
00000290 00 00 00 09 00 23 00 24 00 01 00 0D 00 00 00 4E .....#$. ...N
000002a0 00 02 00 02 00 00 00 12 BB 00 01 59 B7 00 25 4C .....Y...%L
000002b0 2B 10 08 B8 00 0E B6 00 26 B1 00 00 00 02 00 16 +.....&.....
000002c0 00 00 00 0E 00 03 00 00 00 09 00 08 00 0A 00 11 .....
000002d0 00 0B 00 17 00 00 00 16 00 02 00 00 00 12 00 2A .....*
000002e0 00 2B 00 00 00 08 00 0A 00 2C 00 22 00 01 00 01 +.....,.."....
000002f0 00 28 00 29 00 01 00 0D 00 00 00 41 00 02 00 02 ·(·).....A....
00000300 00 00 00 09 2A 2B B6 00 2D B5 00 1B B1 00 00 00 .....*+...-.....
00000310 02 00 16 00 00 00 0A 00 02 00 00 00 0E 00 08 00 .....
00000320 0F 00 17 00 00 00 16 00 02 00 00 00 09 00 21 00 .....!..
00000330 22 00 00 00 00 00 09 00 05 00 08 00 01 00 01 00 ".....
00000340 31 00 00 00 02 00 32 1.....2

```

每一行显示 16 个字节，并且使用十六进制显示，因此一行显示 32 个数字。如果直接看十六进制的是一大串数字，肯定没有人能够知道这一大串数字究竟代表什么，因此需要一定的格式进行组织。下面就拿实际的例子来逐个分析 JVM 所规定的格式。

4.2 魔数与版本

本节基于上文测试用例中所使用的 Test.class 字节码文件，带领大家一起详细分析字节码文件的组成格式（一共 10 个组成部分）。

在具体分析之前，先给出一张字节码文件内容的全图，如图 4.1 所示。

```

00000000 CA FE BA BE 00 00 00 32 00 33 07 00 02 01 00 04 .....2.3.....
00000010 54 65 73 74 07 00 04 01 00 10 6A 61 76 61 2F 6C Test.....java/1
00000020 61 6E 67 2F 4F 62 6A 65 63 74 01 00 01 61 01 00 zng/Object...a..
00000030 01 49 01 00 02 73 69 01 00 13 4C 6A 61 76 61 2F .I...si...Ljava/
00000040 6C 61 6E 67 2F 49 6E 74 65 67 65 72 3B 01 00 01 lang/Integer:...
00000050 73 01 00 12 4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 s...Ljava/lang/S
00000060 74 72 69 6E 67 3B 01 00 08 3C 63 6C 69 6E 69 74 tring:...clinit
00000070 3E 01 00 03 28 29 56 01 00 04 43 6F 64 65 0A 00 >...()V...Code...
00000080 0F 00 11 07 00 10 01 00 11 6A 61 76 61 2F 6C 61 .....java/la
00000090 6E 67 2F 49 6E 74 65 67 65 72 0C 00 12 00 13 01 ng/Integer.....
000000a0 00 07 76 61 6C 75 65 4F 66 01 00 16 29 49 29 4C ..valueOf...()L
000000b0 6A 61 76 61 2F 6C 61 6E 67 2F 49 6E 74 65 67 65 java/lang/Intege
000000c0 72 3B 09 00 01 00 15 0C 00 07 00 08 01 00 0F 4C r:...Integer...L
000000d0 69 6E 65 4E 75 6D 62 65 72 54 61 62 6C 65 01 00 ineNumberTable...
000000e0 12 4C 6F 63 61 6C 56 61 72 69 61 62 6C 65 54 61 .LocalVariableTa
000000f0 62 6C 65 01 00 06 3C 69 6E 69 74 3E 0A 00 03 00 ble...<init>....
00000100 1A 0C 00 18 00 0C 09 00 01 00 1C 0C 00 05 00 06 .....
00000110 08 00 1E 01 00 0C 48 65 6C 6C 6F 20 57 6F 72 6C .....Hello Worl
00000120 64 21 09 00 01 00 20 0C 00 09 00 0A 01 00 04 74 d!.....t
00000130 68 69 73 01 00 06 4C 54 65 73 74 3B 01 00 04 6D his...LTest;...m
00000140 61 69 6E 01 00 16 28 5B 4C 6A 61 76 61 2F 6C 61 ain...([Ljava/la
00000150 6E 67 2F 53 74 72 69 6E 67 3B 29 56 0A 00 01 00 ng/String;)V...
00000160 1A 0A 00 01 00 27 0C 00 28 00 29 01 00 04 73 65 .....'.()..se
00000170 74 41 01 00 16 28 4C 6A 61 76 61 2F 6C 61 6E 67 tA...([Ljava/lang
00000180 2F 49 6E 74 65 67 65 72 3B 29 56 01 00 04 61 72 /Integer;)V...ar
00000190 67 73 01 00 13 5B 4C 6A 61 76 61 2F 6C 61 6E 67 ga...([Ljava/lang
000001a0 2F 53 74 72 69 6E 67 3B 01 00 04 74 65 73 74 0A /String;...test.
000001b0 00 0F 00 2E 0C 00 2F 00 30 01 00 08 69 6E 74 56 ...../0...intV
000001c0 61 6C 75 65 01 00 03 28 29 49 01 00 0A 53 6F 75 alue...()I...Sou
000001d0 72 63 65 46 69 6C 65 01 00 09 54 65 73 74 2E 6A rceFile...Test.g
000001e0 61 76 61 00 21 00 01 00 03 00 00 00 03 00 01 00 ava!.....
000001f0 05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00 .....
00000200 09 00 0A 00 00 00 04 00 08 00 08 00 0C 00 01 00 .....
00000210 0D 00 00 00 29 00 04 00 00 00 00 00 09 10 06 B8 (...)...
00000220 00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 06 .....
00000230 00 01 00 00 00 03 00 17 00 00 00 02 00 00 01 .....
00000240 00 18 00 0C 00 01 00 0D 00 00 00 46 00 02 00 01 .....F...
00000250 00 00 00 10 2A B7 00 19 2A 06 B5 00 1B 2A 12 1D .....*...*...*...
00000260 B5 00 1F B1 00 00 00 02 00 16 00 00 00 12 00 04 .....
00000270 00 00 00 01 00 04 00 02 00 09 00 04 00 0F 00 01 .....
00000280 00 17 00 00 00 0C 00 01 00 00 00 10 00 21 00 22 .....1...
00000290 00 00 00 09 00 23 00 24 00 01 00 0D 00 00 00 4E .....$.S...N
000002a0 00 02 00 02 00 00 00 12 BB 00 01 59 B7 00 25 4C .....Y...$L
000002b0 2B 10 08 B8 00 0E B6 00 26 B1 00 00 00 02 00 16 +.....s.....
000002c0 00 00 00 0E 00 03 00 00 00 07 00 08 00 08 00 11 .....
000002d0 00 09 00 17 00 00 00 16 00 02 00 00 00 12 00 2A .....*
000002e0 00 2B 00 00 00 08 00 0A 00 2C 00 22 00 01 00 01 +.....T...
000002f0 00 28 00 29 00 01 00 0D 00 00 00 41 00 02 00 02 (.).A...
00000300 00 00 00 09 2A 2B B6 00 2D B5 00 1B B1 00 00 00 .....*+...-...
00000310 02 00 16 00 00 00 0A 00 02 00 00 00 0C 00 08 00 .....
00000320 0D 00 17 00 00 00 16 00 02 00 00 00 09 00 21 00 .....!...
00000330 22 00 00 00 00 00 09 00 05 00 08 00 01 00 01 00 "......
00000340 31 00 00 00 02 00 32 1.....2

```

图 4.1 总览 Java 字节码格式

下面会基于图 4.1 截出若干小图片，以分析字节码文件内容的格式。但是为了节省篇幅，每次仅仅截其中一部分，读者可以根据每一行左边的行号进行定位。同时，为了使不同数据结构的前后排列布局显示得更加清晰，下面的截图会将上下几行信息截下来，这样读者能够对整体字节码结构做到心中有数。

4.2.1 魔数

所有.class 字节码文件的开始 4 个字节都是魔数，并且其值一定是 0xCAFEBABE。注意，这里的 CAFEBABE 是指十六进制数值，并不是字符串“CAFEBABE”，如图 4.2 所示。如果开始 4 字节不是 0xCAFEBABE，则 JVM 将会认为该文件不是.class 字节码文件，并拒绝解析。

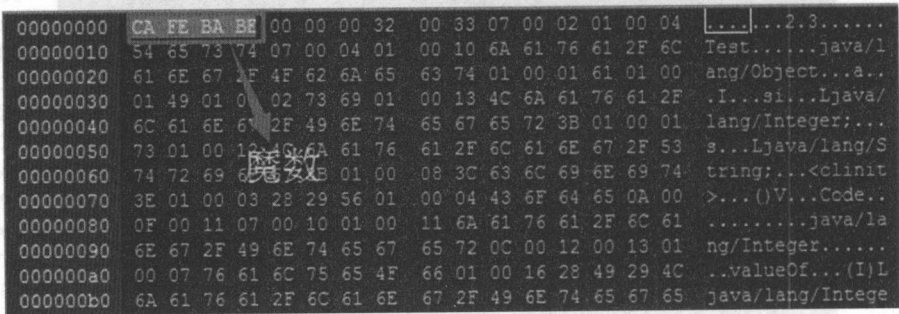


图 4.2 Java 字节码中的魔数

4.2.2 版本号

根据字节码文件规范，魔数之后的 4 个字节为版本信息，前两个字节表示 major version，即主版本号；后两个字节表示 minor version，即次版本号。这里版本号的值为 0x00000032，对应的十进制数是 50。

目前已发布的 version 包括：1.1(45)、1.2(46)、1.3(47)、1.4(48)、1.5(49)、1.6(50)、1.7(51)。据此可以知道，该 class 文件是 JDK 1.6 编译的，如图 4.3 所示。



图 4.3 Java 字节码文件中的版本号

4.3 常量池

常量池是.class 字节码文件中非常重要和核心的内容，一个 Java 类中绝大多数的信息都由常量池描述，尤其是 Java 类中定义的变量和方法，都由常量池保存。注意，对 JVM 有所研究的人，可能都知道 JVM 的内存模型中，有一块就是常量池，JVM 堆区的常量池就是用于保存每一个 Java 类所对应的常量池的信息的，一个 Java 应用程序中所包含的所有 Java 类的常量池，组成了 JVM 堆区中大的常量池。

4.3.1 常量池的基本结构

Java 类所对应的常量池主要由常量池数量和常量池数组两部分组成（如图 4.4 所示），常量池数量紧跟在次版本号后面，占 2 字节。常量池数组则紧跟在常量池数量之后。

常量池数组，顾名思义，就是一个类似数组的结构。这个数组固化在字节码文件中，由多个元素组成。与一般数组概念不同的是，常量池数组中不同的元素的类型、结构都是不同的，长度也是不同的，但是每一种元素的第一个数据都是一个 u1 类型，该字节是标志位，占 1 个字节（如图 4.4 所示）。JVM 解析常量池时，根据这个 u1 类型来获取该元素的具体类型。常量池的结构组成如图 4.4 所示。

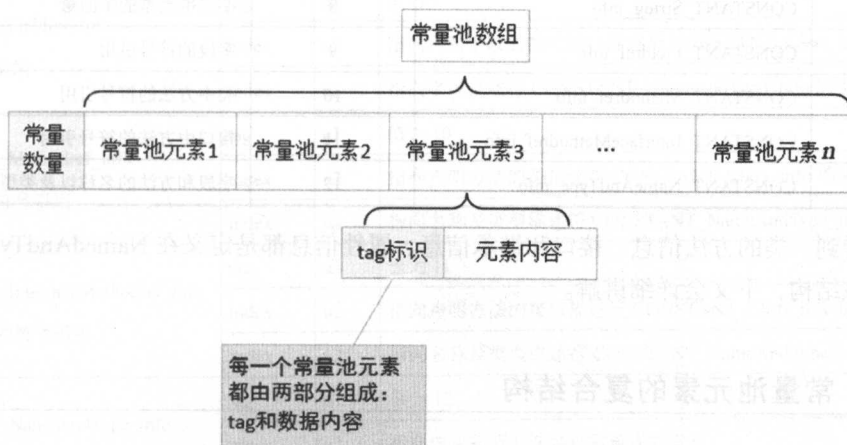


图 4.4 Java 字节码中常量池的组成结构

使用结构化的方式来描述常量池数组的编排，可以如下描述：

tag1 元素内容 1 **tag2** 元素内容 2 **tag3** 元素内容 3 ... **tagn** 元素内容 n

这里为了描述，在 tag 与元素内容之间添加了空格，但实际的 class 二进制文件中，这些 tag 与元素内容之间绝没有任何空格信息，也没有任何其他的多余信息，tag 后面紧跟着元素内容，第一个元素内容结束了，紧接着就是第二个元素的 tag 信息，由此可见，整个字节码文件的格式设计都是非常紧凑的。

4.3.2 JVM 所定义的 11 种常量

常量池元素中的不同元素结构与类型都是不同的，正因如此，JVM 只能定义有限的元素类型，并针对有限的类型进行专门解析。JVM 一共定义了 11 种常量，如表 4.1 所示。

表 4.1 JVM 常量池元素一览表

编 号	常量池元素名称	tag 位标识	含 义
1	CONSTANT_Utf8_info	1	UTF-8 编码的字符串
2	CONSTANT_Integer_info	3	整型字面量
3	CONSTANT_Float_info	4	浮点型字面量
4	CONSTANT_Long_info	5	长整型字面量
5	CONSTANT_Double_info	6	双精度字面量
6	CONSTANT_Class_info	7	类或接口的符号引用
7	CONSTANT_String_info	8	字符串类型的字面量
8	CONSTANT_Fieldref_info	9	字段的符号引用
9	CONSTANT_Methodref_info	10	类中方法的符号引用
10	CONSTANT_InterfaceMethodref_info	11	接口中方法的符号引用
11	CONSTANT_NameAndType_info	12	字段和方法的名称以及类型的符号引用

可以看到，类的方法信息、接口和继承信息、属性信息都是定义在 NamedAndType_Info 中的，关于该结构，下文会详细讲解。

4.3.3 常量池元素的复合结构

常量池数组中的每一种元素的内容都是复合数据结构的，下面分别给出 JVM 所定义的常量池中每一种元素的具体结构（如表 4.2 所示）。

表 4.2 常量池元素结构

类型/含义	结构	类型	描 述
CONSTANT_Utf8_info UTF-8 编码的字符串	tag	u1	值为 1
	length	u2	UTF-8 缩略编码字符串占用字节数
	bytes	u1	长度为 length 的 UTF-8 编码字符串
CONSTANT_Integer_info 整型字面量	tag	u1	值为 3
	bytes	u4	按照高位在前储存的 int 值
CONSTANT_Float_info 浮点型字面量	tag	u1	值为 4
	bytes	u4	按照高位在前储存的 float 值
CONSTANT_Long_info 长整型字面量	tag	u1	值为 5
	bytes	u8	按照高位在前储存的 long 值
CONSTANT_Double_info 双精度浮点型字面量	tag	u1	值为 6
	bytes	u8	按照高位在前储存的 double 值
CONSTANT_Class_info 类或接口的符号引用	tag	u1	值为 7
	index	u2	指向全限定名常量项的索引
CONSTANT_String_info 字符串类型字面量	tag	u1	值为 8
	index	u2	指向字符串字面量的索引
CONSTANT_Fieldref_info 字段的符号引用	tag	u1	值为 9
	index	u2	指向声明字段的类或接口描述符 CONSTANT_Class_info 的索引项
	index	u2	指向字段描述符 CONSTANT_NameAndType_info 的索引项
CONSTANT_Methodref_info 类中方法的符号引用	tag	u1	值为 10
	index	u2	指向声明方法的类描述符 CONSTANT_Class_info 的索引项
	index	u2	指向名称及类型描述符 CONSTANT_NameAndType_info 的索引项
CONSTANT_InterfaceMethodref_info 接口中方法的符号引用	tag	u1	值为 11
	index	u2	指向声明方法的接口描述符 CONSTANT_Class_info 的索引项
	index	u2	指向名称及类型描述符 CONSTANT_NameAndType_info 的索引项
CONSTANT_NameAndType_info 字段或方法的部分符号引用	tag	u1	值为 12
	index	u2	指向该字段或方法名称常量项的索引
	index	u2	指向该字段或方法描述符常量项的索引

该表中 tag 值为 1 的常量池元素 `CONSTANT_Utf8_info`，其组成结构分为 3 部分，分别是：tag、length 和 bytes，其中 tag 和 length 的长度分别是 u1、u2，即分别占 1 字节和 2 字节。而 bytes 则是字符串的具体内容，其长度是 length 字节。在字节码文件中，该常量池元素最终所占的字节数是：

$$1 + 2 + \text{length}$$

其他类型的常量池元素的组成结构类似，这里不一一分析，下文在实例解析常量池组成结构时会再次逐个进行详细解析。

4.3.4 常量池的结束位置

相信有不少读者读到这里，可能潜意识里会突然蹦出这么一个问题：整个字节码文件由多个部分构成，常量池数组只是其中一块，JVM 在解析字节码文件时，一定需要分别读取各个部分的字节流，其中也包括常量池数组。但是常量池中有部分元素的值是 bytes 数组，其长度是随机变化的，那么 JVM 在解析时，是如何知道整个常量池的信息解析到什么位置结束呢？其实经过分析不难发现，首先，class 文件给出了常量池的总数；其次，凡是碰到有 bytes 数组的常量池元素，class 文件在常量池的每一个元素之前都会专门划分出 2 字节用于描述该常量池元素内容所占的字节长度，这样一来，常量池中每一个元素的长度是确定的，而常量池的总数也是确定的，JVM 据此便可以从 class 文件中准确地计算出常量池结构体的末端位置（起始位置不用计算，是定死的，从第 9 字节开始，前面 8 字节分别是魔数和版本号）。

对上面这些概念有了形象上的认识后，下面让我们一起继续以 `Test.class` 这个字节码文件为例，详细分析其中的常量池信息。

4.3.5 常量池元素总数量

前面 8 字节用于描述魔数和版本号，从第 9 字节开始的一大段字节流都用于描述常量池数组信息。其中，第 9 和第 10 字节用于描述常量池元素的总数量。如图 4.5 所示。

第 9 和第 10 字节所保存的常量池数组大小是 `0x33`，换算成十进制是 51，说明该字节码文件中一共包含 51 个常量池元素。JVM 规定，不使用第 0 个元素，因此实际上一共有 50 个常量池元素（下文在解析源码时，会看到源码中的确是从第 1 个元素开始解析的，而不是从第 0 个）。

00000000	CA FE BA BE 00 00 00 32	00 33 07 00 02 01 00 042.3.....
00000010	54 65 73 74 07 00 04 01	00 10 6A 61 76 61 2F 6C	Test.....java/l
00000020	61 6E 67 2F 4F 62 6A 65	63 74 01 00 01 61 01 00	ang/Object...a..
00000030	01 49 01 00 02 73 69 01	00 13 4C 6A 61 76 61 2F	.I...si...Ljava/
00000040	6C 61 6E 67 2F 49 6E 74	65 67 65 72 3B 01 00 01	lang/Integer;...
00000050	73 01 00 12 4C 6A 61 76	61 6E 67 2F 53	s...Ljava/lang/S
00000060	74 72 69 6E 67 3B 01 00	6C 69 6E 69 74	tring;...<clinit
00000070	3E 01 00 03 28 29 56 01	00 04 43 6F 64 65 0A 00	>...()V...Code..
00000080	0F 00 11 07 00 10 01 00	11 6A 61 76 61 2F 6C 61java/la
00000090	6E 67 2F 49 6E 74 65 67	65 72 0C 00 12 00 13 01	ng/Integer.....
000000a0	00 07 76 61 6C 75 65 4F	66 01 00 16 28 49 29 4C	..valueOf...(I)L
000000b0	6A 61 76 61 2F 6C 61 6E	67 2F 49 6E 74 65 67 65	java/lang/Intege

图 4.5 常量池长度

4.3.6 第一个常量池元素

常量池数量之后（即从第 11 字节开始），就是常量池数组。每一个常量池元素都以 tag 位标开始，tag 位标都只占 1 字节长度。如图 4.6 所示。第 11 字节对应的值是 7，对照上文所给的常量池 11 种元素的复合结构可知，tag 位标为 7 代表的是 CONSTANT_Class_info，即类或接口的符号引用，这种类型的元素的结构组成如下：

- ◎ tag 位标 占 1 字节
- ◎ index 占 2 字节

既然 tag 位标已经占据了字节码文件的第 11 字节，则接下来的第 12 和 13 字节将合起来表示 index。如图 4.6 所示，这两个字节对应的值为 2。

从第 11 字节开始，到第 13 字节为止，一个常量池元素就被描述完成。紧接着开始描述第 2 个常量池元素。

00000000	CA FE BA BE 00 00 00 32	00 33 07 00 02 01 00 042.3.....
00000010	54 65 73 74 07 00 04 01	00 10 6A 61 76 61 2F 6C	Test.....java/l
00000020	61 6E 67 2F 4F 62 6A 65	63 74 01 00 01 61 01 00	ang/Object...a..
00000030	01 49 01 00 02 73 69 01	00 13 4C 6A 61 76 61 2F	.I...si...Ljava/
00000040	6C 61 6E 67 2F 49 6E 74	65 67 65 72 3B 01 00 01	lang/Integer;...
00000050	73 01 00 12 4C 6A 61 76	61 6E 67 2F 53	s...Ljava/lang/S
00000060	74 72 69 6E 67 3B 01 00	6C 69 6E 69 74	tring;...<clinit
00000070	3E 01 00 03 28 29 56 01	00 04 43 6F 64 65 0A 00	>...()V...Code..
00000080	0F 00 11 07 00 10 01 00	11 6A 61 76 61 2F 6C 61java/la
00000090	6E 67 2F 49 6E 74 65 67	65 72 0C 00 12 00 13 01	ng/Integer.....
000000a0	00 07 76 61 6C 75 65 4F	66 01 00 16 28 49 29 4C	..valueOf...(I)L
000000b0	6A 61 76 61 2F 6C 61 6E	67 2F 49 6E 74 65 67 65	java/lang/Intege
000000c0	72 3B 09 00 01 00 15 0C	00 07 00 08 01 00 0F 4C	ry;.....L

图 4.6 常量池第一个元素的 tag 和 index

4.3.7 第二个常量池元素

第一个常量池元素后面紧跟着的是第二个常量池元素。其第一字节是 01，表示这是一个 UTF8 编码的字符串，其结构是：

- ◎ tag 位，占 1 字节
- ◎ length，占 2 字节
- ◎ bytes，占 length 字节

如图 4.7 所示，tag 位后面的 2 字节的值是 4，表示 bytes 占 4 字节，其值为 0x54657374，其中每两字节正好代表一个字符，对应的字符串是 Test。

00000000	CA	FE	BA	BE	00	00	00	32	00	33	07	00	02	01	00	042.3...[...]
00000010	54	65	73	74	07	00	04	01	00	10	6A	61	76	61	2F	6C	Test[...].java/1
00000020	61	6E	67	2F	4F	62	6A	65	63	74	01	00	01	61	01	00	ang/Object...a..
00000030	01	49	01	00	02	73	69	01	00	13	4C	6A	61	76	61	2F	.I...si...Ljava/
00000040	6C	61	6E	67	2F	49	6E	74	65	67	65	72	3B	01	00	01	lang/Integer;...
00000050	73	01	00	12	4C	6A	61	76	61	2F	6C	61	6E	67	2F	53	s...Ljava/lang/S
00000060	74	72	69	6E	67	3B	01	00	08	3C	63	6C	69	6E	69	74	tring;...<clinit
00000070	3E	01	00	03	28	29	56	01	00	04	43	6F	64	65	0A	00	>...()V...Code..
00000080	0F	00	11	07	00	10	01	00	11	6A	61	76	61	2F	6C	61java/la
00000090	6E	67	2F	49	6E	74	65	67	65	72	0C	00	12	00	13	01	ng/Integer.....

图 4.7 常量池第二个元素

4.3.8 父类常量

刚才的第 1 与第 2 两个常量用于描述 Java 类型信息，接下来的第 3 与第 4 两个常量则用于描述父类信息。由于 Test.Java 类并没有显式继承任何类，因此编译后处理成默认继承，即父类是 Java.lang.Object。

父类常量的 tag 位也是 07（如图 4.8 所示），其类名是 java/lang/Object（如图 4.9 所示）。下面分别给出第 3 和第 4 这两个常量在文件中的内容。

00000000	CA	FE	BA	BE	00	00	00	32	00	33	07	00	02	01	00	042.3.....
00000010	54	65	73	74	07	00	04	01	00	10	6A	61	76	61	2F	6C	Test[...].java/1
00000020	61	6E	67	2F	4F	62	6A	65	63	74	01	00	01	61	01	00	ang/Object...a..
00000030	01	49	01	00	02	73	69	01	00	13	4C	6A	61	76	61	2F	.I...si...Ljava/
00000040	6C	61	6E	67	2F	49	6E	74	65	67	65	72	3B	01	00	01	lang/Integer;...
00000050	73	01	00	12	4C	6A	61	76	61	2F	6C	61	6E	67	2F	53	s...Ljava/lang/S
00000060	74	72	69	6E	67	3B	01	00	08	3C	63	6C	69	6E	69	74	tring;...<clinit
00000070	3E	01	00	03	28	29	56	01	00	04	43	6F	64	65	0A	00	>...()V...Code..
00000080	0F	00	11	07	00	10	01	00	11	6A	61	76	61	2F	6C	61java/la
00000090	6E	67	2F	49	6E	74	65	67	65	72	0C	00	12	00	13	01	ng/Integer.....

图 4.8 第 3 个常量——父类信息

00000000	CA FE BA BE 00 00 00 32	00 33 07 00 02 01 00 042.3.....
00000010	54 65 73 74 07 00 04 01	00 10 6A 61 76 61 2F 6C	Test.....java/1
00000020	61 6E 67 2F 4F 62 6A 65	63 74 01 00 01 61 01 00	ang/Object.....a..
00000030	01 49 01 00 02 73 69 01	00 13 4C 6A 61 76 61 2F	.I...si...Ljava/
00000040	6C 61 6E 67 2F 49 6E 74	65 67 65 72 3B 01 00 01	lang/Integer;...
00000050	73 01 00 12 4C 6A 61 76	61 2F 6C 61 6E 67 2F 53	s...Ljava/lang/S
00000060	74 72 69 6E 67 3B 01 00	08 3C 63 6C 69 6E 69 74	tring;...<clinit
00000070	3E 01 00 03 28 29 56 01	00 04 43 6F 64 65 0A 00	>...()V...Code..
00000080	0F 00 11 07 00 10 01 00	11 6A 61 76 61 2F 6C 61java/la
00000090	6E 67 2F 49 6E 74 65 67	65 72 0C 00 12 00 13 01	ng/Integer.....

图 4.9 第 4 个常量——父类接口名称

图 4.9 显示，字符串的 length 值为 0x10，即 16，于是其后面的 16 字节都是 bytes。由此可以进一步验证，字符串常量的结构由 tag、length 和 bytes 组成，bytes 的长度由 length 指定。

4.3.9 变量型常量池元素

常量池中前面 4 个元素主要用于描述 Java 类自身的名称和其父类名称，接下来的字节码流则开始描述类中的变量信息，这些变量既包括类的成员变量，也包括类变量（即静态变量）信息。

首先看类成员变量 a 的信息，如图 4.10 所示。

00000000	CA FE BA BE 00 00 00 32	00 33 07 00 02 01 00 042.3.....
00000010	54 65 73 74 07 00 04 01	00 10 6A 61 76 61 2F 6C	Test.....java/1
00000020	61 6E 67 2F 4F 62 6A 65	63 74 01 00 01 61 01 00	ang/Object.....a..
00000030	01 49 01 00 02 73 69 01	00 13 4C 6A 61 76 61 2F	.I...si...Ljava/
00000040	6C 61 6E 67 2F 49 6E 74	65 67 65 72 3B 01 00 01	lang/Integer;...
00000050	73 01 00 12 4C 6A 61 76	61 2F 6C 61 6E 67 2F 53	s...Ljava/lang/S
00000060	74 72 69 6E 67 3B 01 00	08 3C 63 6C 69 6E 69 74	tring;...<clinit
00000070	3E 01 00 03 28 29 56 01	00 04 43 6F 64 65 0A 00	>...()V...Code..
00000080	0F 00 11 07 00 10 01 00	11 6A 61 76 61 2F 6C 61java/la
00000090	6E 67 2F 49 6E 74 65 67	65 72 0C 00 12 00 13 01	ng/Integer.....

图 4.10 变量 a 所对应的常量池信息

图 4.10 中所选中的 8 字节，一共包含两个常量池元素信息，这两个常量池元素的类型都是字符串，因为其 tag 位都是 1。第一个字符串常量的 length 是 1，其值（即 bytes）是 0x61，正好对应 UTF-8 编码的字符 a。第二个字符串常量的 length 也是 1，其值是 0x49，正好对应 UTF-8 编码的字符 I。在 JVM 规范中，若变量的类型是 I，则表示该变量的实际类型是 int。这与上文对变量 a 的定义一致。

接着看类变量 si 的定义，如图 4.11 所示。

00000000	CA FE BA BE 00 00 00 32	00 33 07 00 02 01 00 042.3.....
00000010	54 65 73 74 07 00 04 01	00 10 6A 61 76 61 2F 6C	Test.....java/l
00000020	61 6E 67 2F 4F 62 6A 65	63 74 01 00 01 61 01 00	ang/Object...a..
00000030	01 49 01 00 02 73 69 01	00 13 4C 6A 61 76 61 2F	.I...si...Ljava/
00000040	6C 61 6E 67 2F 49 6E 74	65 67 65 72 3E 01 00 01	lang/Integer;...
00000050	73 01 00 12 4C 6A 61 76	61 2F 6C 61 6E 67 2F 53	s...Ljava/lang/S
00000060	74 72 69 6E 67 3B 01 00	08 3C 63 6C 69 6E 69 74	tring;...<clinit
00000070	3E 01 00 03 28 29 56 01	00 04 43 6F 64 65 0A 00	>...()V...Code..
00000080	0F 00 11 07 00 10 01 00	11 6A 61 76 61 2F 6C 61java/la
00000090	6E 67 2F 49 6E 74 65 67	65 72 0C 00 12 00 13 01	ng/Integer.....

图 4.11 变量 si 所对应的常量池信息

图 4.11 中所选中的 27 字节，一共描述了两个常量池元素，这两个常量池元素的类型也都是字符串。第一个字符串的 length 为 2，其值是 0x7369，对应 utf-8 编码的字符串 si。第二个字符串的 length 为 0x13，即 19，其值是 0x4C 6A 61 76 61 2F 6C 61 6E 67 2F 49 6E 74 65 67 65 72 3E，这一串值是 ASCII 字符，每 2 个十六进制数对应一个 ASCII 字符，这些数字连起来就对应一个字符串，所对应的字符串是 Ljava/lang/Integer;。

这两个常量池元素合起来，描述了 Test 类中的 static Integer si 这样的类变量。

接着看类成员变量 s 的定义，如图 4.12 所示。

00000000	CA FE BA BE 00 00 00 32	00 33 07 00 02 01 00 042.3.....
00000010	54 65 73 74 07 00 04 01	00 10 6A 61 76 61 2F 6C	Test.....java/l
00000020	61 6E 67 2F 4F 62 6A 65	63 74 01 00 01 61 01 00	ang/Object...a..
00000030	01 49 01 00 02 73 69 01	00 13 4C 6A 61 76 61 2F	.I...si...Ljava/
00000040	6C 61 6E 67 2F 49 6E 74	65 67 65 72 3B 01 00 01	lang/Integer;...
00000050	73 01 00 12 4C 6A 61 76	61 2F 6C 61 6E 67 2F 53	s...Ljava/lang/S
00000060	74 72 69 6E 67 3E 01 00	08 3C 63 6C 69 6E 69 74	tring;...<clinit
00000070	3E 01 00 03 28 29 56 01	00 04 43 6F 64 65 0A 00	>...()V...Code..
00000080	0F 00 11 07 00 10 01 00	11 6A 61 76 61 2F 6C 61java/la
00000090	6E 67 2F 49 6E 74 65 67	65 72 0C 00 12 00 13 01	ng/Integer.....

图 4.12 变量 s 所对应的常量池信息

图 4.12 中选中的 25 字节，一共描述了两个常量池元素，这两个常量池元素的类型也都是字符串。第一个字符串的 length 为 1，其值是 0x73，对应 UTF-8 编码的字符 s。第二个字符串的 length 为 0x12，即 18，其值是 0x4C 6A 61 76 61 2F 6C 61 6E 67 2F 53 74 72 69 6E 67 3B，对应 UTF-8 编码的字符串 Ljava/lang/String;。

这两个常量池元素合起来描述了 Test 类中的 s 字符串变量。

4.4 访问标识与继承信息

上面解析完了常量池的所有元素，可能读者已经陷入常量池的分析之中，但是别忘了，一个.class 字节码文件中共有 10 个组成部分（见上文），常量池只不过是其中的一个组成部分，因此现在需要把目光转移出来，继续回到.class 字节码文件后续的分析。

4.4.1 access_flags

在字节码文件中，常量池数组之后紧接着的是 access_flags 结构，该结构类型是 u2，占 2 字节。

access_flags 代表访问标志位，该标志用于标注类或接口层次的访问信息，如当前 Class 是类还是接口，是否定义为 public 类型，是否定义为 abstract 类型等。

对于本文中的测试用例 Test.class，其 access_flags 信息如图 4.13 所示。

图 4.13 access 属性

由图 4.13 可知，Test.class 的 access_flags 的值是 0x0021，这代表什么含义呢？根据 JVM 的规范，access_flags 的可选项值如表 4.3 所示。

表 4.3 access_flags 可选项

标志名称	标志值	含 义
ACC_PUBLIC	0x0001	是否为 public 类型
ACC_FINAL	0x0010	是否被声明为 final，只有类可设置
ACC_SUPER	0x0020	是否允许使用 invokespecial 字节码指令，JDK1.2 以后编译出来的类这个标志为真
ACC_INTERFACE	0x0200	标识这是一个接口

续表

标志名称	标志值	含 义
ACC_ABSTRACT	0x0400	是否为 abstract 类型，对于接口和抽象类，此标志为真，其他类为假
ACC_SYNTHETIC	0x1000	标识这个类并非由用户代码产生
ACC_ANNOTATION	0x2000	标识这是一个注解
ACC_ENUM	0x4000	标识这是一个枚举

由于 Test.class 中的 access_flags=0x21, 因此该类的访问标识既包含 ACC_PUBLIC(0x0001), 也包含 ACC_SUPER(0x0020)。其中, 自 JDK1.2 以后, 类被编译出来的 invokespecial 字节码指令是否允许使用的选项都是真, 因此 access_flags 的值都会带有 ACC_SUPER 标识位。

4.4.2 this_class

在字节码文件中, 紧跟着 access_flags 访问标识之后的是 this_class 结构, 该结构类型是 u2, 占 2 字节。this_class 记录当前类的全限定名 (包名+类名), 其值指向常量池中对应的索引值。

Test.class 中的 this_class 信息如图 4.14 所示。

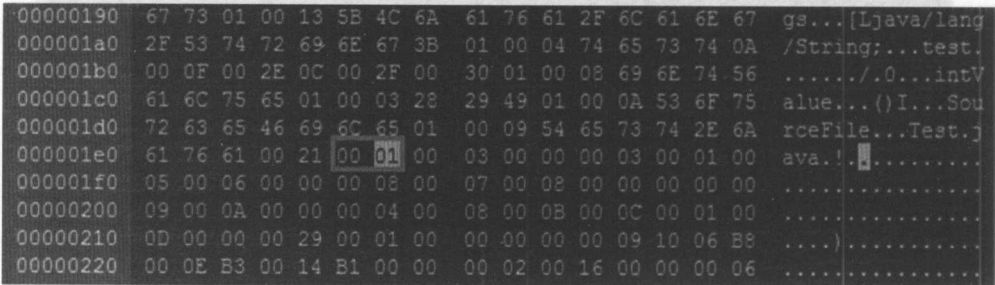


图 4.14 this_class 的字节码

由图 4.14 可知, Test.class 的 this_class 的值为 1, 说明该值对应 1 号常量池元素。上文使用 javap -verbose 命令将 Test.class 的所有常量池连同其编号一起打印了出来, 根据打印信息可知, 1 号常量池元素的信息如下:

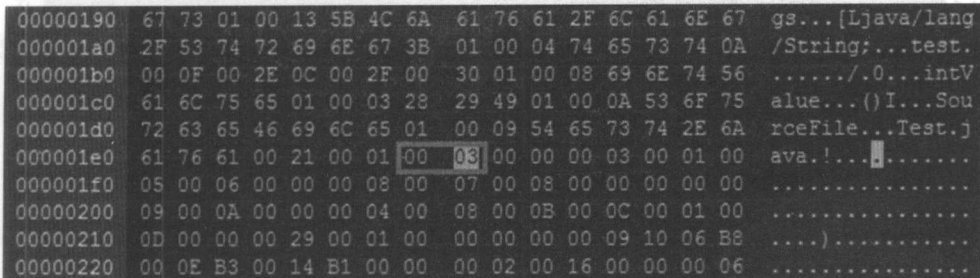
```
const #1 = class      #2;    // Test
const #2 = Asciz      Test;
```

由此可知, this_class 的确是 Test, 类的全限定名就是 Test。

4.4.3 super_class

在字节码文件中，紧跟着 this_class 访问标识之后的是 super_class 结构，该结构类型是 u2，占 2 字节。super_class 记录当前类的父类全限定名，其值指向常量池中对应的索引值。

Test.class 中的 super_class 信息如图 4.15 所示。



```

00000190  67 73 01 00 13 5b 4c 6a 61 76 61 2f 6c 61 6e 67  gs...[Ljava/lang
000001a0  2f 53 74 72 69 6e 67 3b 01 00 04 74 65 73 74 0a  /String;...test.
000001b0  00 0f 00 2e 0c 00 2f 00 30 01 00 08 69 6e 74 56  ...../.0...intV
000001c0  61 6c 75 65 01 00 03 28 29 49 01 00 0a 53 6f 75  alue...()I...Sou
000001d0  72 63 65 46 69 6c 65 01 00 09 54 65 73 74 2e 6a  rceFile...Test.j
000001e0  61 76 61 00 21 00 01 00 03 00 00 00 03 00 01 00  ava.!...
000001f0  05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00  .....
00000200  09 00 0a 00 00 00 04 00 08 00 0b 00 0c 00 01 00  .....
00000210  0d 00 00 00 29 00 01 00 00 00 00 00 09 10 06 b8  ....).....
00000220  00 0e b3 00 14 b1 00 00 00 02 00 16 00 00 00 06  .....

```

图 4.15 super_class 的字节码

由图 4.15 可知，Test.class 的 super_class 的值为 3，说明该值对应 3 号常量池元素。上文使用 javap -verbose 命令将 Test.class 的所有常量池连同其编号一起打印了出来，根据打印信息可知，3 号常量池元素的信息如下：

```

const #3 = class      #4;    // java/lang/Object
const #4 = Asciz      java/lang/Object;

```

由于 Test.class 并没有显式继承任何基类，因此编译时便让其默认继承 java.lang.Object。这与字节码中的 super_class 值是一致的。

4.4.4 interface

1. interfaces_count

在字节码文件中，紧跟着 super_class 访问标识之后的是 interfaces_count 结构，该结构类型是 u2，占 2 字节。interfaces_count 结构记录当前类所实现的接口数量。

Test.class 中的 interfaces_count 结构信息如图 4.16 所示。

```

00000190 67 73 01 00 13 5B 4C 6A 61 76 61 2F 6C 61 6E 67 gs...[Ljava/lang
000001a0 2F 53 74 72 69 6E 67 3B 01 00 04 74 65 73 74 0A /String;...test.
000001b0 00 0F 00 2E 0C 00 2F 00 30 01 00 08 69 6E 74 56 ...../.0...intV
000001c0 61 6C 75 65 01 00 03 28 29 49 01 00 0A 53 6F 75 alue...()I...Sou
000001d0 72 63 65 46 69 6C 65 01 00 09 54 65 73 74 2E 6A rceFile...Test.j
000001e0 61 76 61 00 21 00 01 00 03 00 00 00 03 00 01 00 ava.!.....
000001f0 05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00 .....
00000200 09 00 0A 00 00 00 04 00 08 00 0B 00 0C 00 01 00 .....
00000210 0D 00 00 00 29 00 01 00 00 00 00 00 09 10 06 B8 ....).....
00000220 00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 06 .....

```

图 4.16 字节码中的接口信息

由图 4.16 可知, Test.class 的 `interfaces_count` 的值为 0, 说明 Test.class 并没有实现任何接口。

2. `interfaces[interfaces_count]`

`interfaces` 表示接口索引集合, 是一组 `u2` 类型数据的集合, 该结构描述当前类实现了哪些接口, 这些被实现的接口将按 `implements` 语句 (如果该类本身为接口, 则为 `extends` 语句) 后的接口顺序从左至右排列在接口的索引集合中。

由于 Test.class 的 `interfaces_count` 值为 0, 因此字节码文件中并没有 `interfaces` 信息。

4.5 字段信息

4.5.1 `fields_count`

在字节码文件中, 接口区之后紧接着是 `fields_count` 结构。该结构类型是 `u2`, 占 2 字节。该值记录当前类中所定义的变量总数量, 包括类成员变量和类变量 (即静态变量)。

Test.class 中的 `fields_count` 信息如图 4.17 所示。

```

00000190 67 73 01 00 13 5B 4C 6A 61 76 61 2F 6C 61 6E 67 gs...[Ljava/lang
000001a0 2F 53 74 72 69 6E 67 3B 01 00 04 74 65 73 74 0A /String;...test.
000001b0 00 0F 00 2E 0C 00 2F 00 30 01 00 08 69 6E 74 56 ...../.0...intV
000001c0 61 6C 75 65 01 00 03 28 29 49 01 00 0A 53 6F 75 alue...()I...Sou
000001d0 72 63 65 46 69 6C 65 01 00 09 54 65 73 74 2E 6A rceFile...Test.j
000001e0 61 76 61 00 21 00 01 00 03 00 00 00 03 00 01 00 ava.!.....
000001f0 05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00 .....
00000200 09 00 0A 00 00 00 04 00 08 00 0B 00 0C 00 01 00 .....
00000210 0D 00 00 00 29 00 01 00 00 00 00 00 09 10 06 B8 ....).....
00000220 00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 06 .....

```

图 4.17 字节码文件中的字段数量

由图 4.17 可知, Test.class 类中一共包含 3 个变量。从 Test 源文件也可以看出, 该类的确包含 3 个变量, 分别是 a、si 和 s。

4.5.2 field_info fields[fields_count]

在字节码文件中, 紧跟着 fields_count 之后的是 fields 结构, 该结构长度不确定, 不同的变量类型所占长度是不同的。fields 记录类中所定义的各个变量的详细信息, 包括变量名、变量类型、访问标识、属性等。

1. fields 结构组成格式

要分析 fields 结构信息, 首先需要清楚该结构的数据组成格式, 其格式如表 4.4 所示。

表 4.4 fields 结构组成

类 型	名 称	数 量
u2	access_flags	1
u2	name_index	1
u2	descriptor_index	1
u2	attributes_count	1
attribute_info	attributes	attributes_count

表 4.3 中各个组成元素的说明如下:

- ◎ access_flags, 标识变量的访问标识, 该值是可选的, 由 JVM 规范规定。
- ◎ name_index, 表示变量的简单名称引用, 占 2 字节, 其值指向常量池的索引。
- ◎ descriptor_index, 表示变量的类型信息引用, 占 2 字节, 其值指向常量池的索引。

fields 结构体实际上是一个数组, 数组中的每一个元素的结构都如表 4.3 所示, 即每一个元素都包含访问标识、名称索引、描述信息索引、属性数量和属性信息。其中, 如果属性数量为 0, 则没有属性信息。由于访问标识、名称、描述信息、属性数量的字节长度是确定的, 因此 JVM 可以在解析过程中计算出 fields 结构所占的全部字节数。

变量的 access_flags 有如表 4.5 所示的可选项。

表 4.5 access_flags 的可选项

标志名称	标志值	含 义
ACC_PUBLIC	0x0001	字段是否为 public
ACC_PRIVATE	0x0002	字段是否为 private
ACC_PROTECTED	0x0004	字段是否为 protected
ACC_STATIC	0x0008	字段是否为 static
ACC_FINAL	0x0010	字段是否为 final
ACC_VOLATILE	0x0040	字段是否为 volatile
ACC_TRANSIENT	0x0080	字段是否为 transient
ACC_SYNTHETIC	0x1000	字段是否为编译器自动产生
ACC_ENUM	0x4000	字段是否为 enum

其中，ACC_PUBLIC、ACC_PRIVATE 和 ACC_PROTECTED 这 3 个标志只能选择一个，接口中的字段必须有 ACC_PUBLIC、ACC_STATIC 和 ACC_FINAL 标志，class 文件对此并无规定，这些都是 Java 语言所要求的。

2. 第 1 个变量 a

分析完理论，下面来看看 Test.class 字节码中的变量信息实际都指的是什么。第一个 field 紧跟在 fields_count 这个结构体之后，如图 4.18 所示。

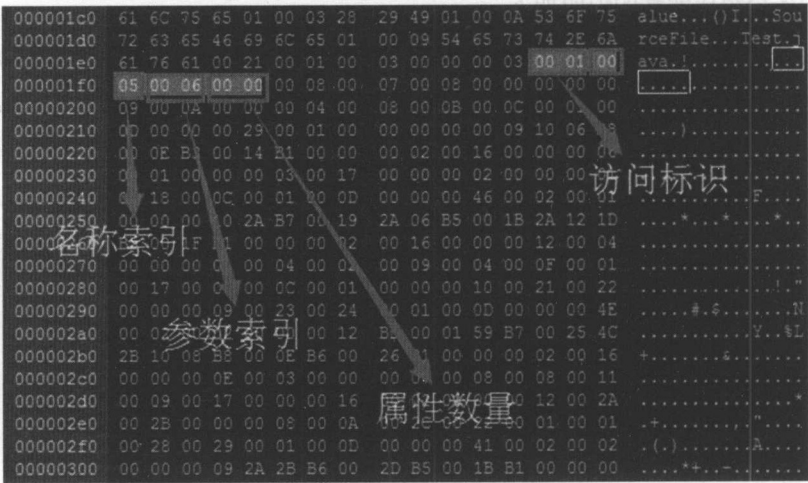


图 4.18 变量 a 在字节码中的描述信息

由图 4.18 可知，第一个变量的访问标识是 1，由参考变量访问标识选项表可知，1 表示 ACC_PUBLIC。第一个变量的名称索引是 5，类型索引是 6，按照上文使用 `javap -verbose` 命令所打印的字节码常量池信息可知，常量池中第 5 和第 6 个元素的信息如下：

```
const #5 = Asciz      a;
const #6 = Asciz      I;
```

由此可知，当前描述的变量是 `a`，其数据类型是 `int`。

根据图 4.18 还可知道，变量 `a` 的属性数量是 0，因为没有属性，所以字段描述结构中最后的元素 `attributes` 也就不存在。

这里需要注意描述信息索引，其指向常量池中的 6 号元素，6 号元素的值是 `I`，这里的 `I` 代表什么意思呢？在 JVM 规范中，每个变量/字段都有描述信息，描述信息主要描述字段的数据类型、方法的参数列表（包括数量、类型和顺序）和返回值。根据描述符规则，基本数据类型和代表无返回值的 `void` 类型都用一个大写字符表示，而对象类型则用字符 `L` 加对象全限定名表示。为了压缩字节码文件的体积（字节码文件最终也会占用服务器硬盘资源和内存资源），对于基本数据类型，JVM 都仅使用一个大写字母来标识。表 4.6 所示是各个基本数据类型所对应的标识符。

表 4.6 标识字符与基本数据类型对应表

标识字符	含 义
B	基本类型 <code>byte</code>
C	基本类型 <code>char</code>
D	基本类型 <code>double</code>
F	基本类型 <code>float</code>
I	基本类型 <code>int</code>
J	基本类型 <code>long</code>
S	基本类型 <code>short</code>
Z	基本类型 <code>boolean</code>
V	特殊类型 <code>void</code>
L	对象类型，如 <code>Ljava/lang/Object</code>

对于数组类型，每一维将使用一个前置的 “[” 字符来描述，如 “`int[]`” 将被记录为 “[I”，“`String[][]`” 将被记录为 “[[Ljava/lang/String;”。

用描述符描述方法时，按照先参数列表，后返回值的顺序描述，参数列表按照参数的严格顺序放在一组 “()” 之内，如方法 “`String getAll(int id,String name)`” 的描述符为

“(Ljava/lang/String;)Ljava/lang/String;”。

3. 变量 si 和 s

由于变量 a 的属性数量是 0，字节码文件中不包含 attributes 信息，因此第一个变量只占 8 字节，分别是访问标识、变量名引用、描述信息引用和属性数量。由于字节码文件标识一共包含 3 个变量，因此描述变量 a 的字节码流的后面的字节码流将继续描述另外两个变量。

Test.class 类的另外两个变量的字节码内容如图 4.19 所示。



图 4.19 变量 si 和 s 在字节码中的描述信息

从图 4.19 可知，这两个变量在字节码文件中也各占 8 字节，因为其 attributes_count 都是 0。变量 si 的 access_flags 是 8，表示这是一个带有 static 修饰符的变量，而变量 s 的 access_flags 是 0，表示该变量没有任何访问修饰符，对照源程序，的确是这样。

两个变量名称分别引用常量池中的 7 号元素和 9 号元素，对照上文使用 javap -verbose 命令打印的常量池信息，可知这两个变量名分别是 si 和 s。

两个变量描述信息分别引用常量池中的 8 号和 10 号元素，对照打印出来的常量池信息可知，其变量类型分别是 Ljava/lang/Integer 和 Ljava/lang/String。由此也可知，对于引用类型的变量，字节码文件描述其变量类型的格式是“L+类全限定名”。

至此，字节码中对 fields 的描述全部结束。不过为了追求简单易懂，本示例中并没有对类变量添加任何属性，因此本示例所生成的字节码的字段信息，都没有 attribute 信息。不过各位道友可以自行做试验进行更加深入的研究。

4.6 方法信息

4.6.1 methods_count

在字节码文件中，紧跟着变量描述结构 `fields` 后面的是 `methods_count` 结构，该结构类型是 `u2`，占 2 字节。该结构描述类中一共包含多少个方法。

`Test.class` 字节码文件中该结构信息如图 4.20 所示。

```

000001d0 72 63 65 46 69 6C 65 01 00 09 54 65 73 74 2E 6A rceFile...Test.]
000001e0 61 76 61 00 21 00 01 00 03 00 00 00 03 00 01 00 ava.!.....
000001f0 05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00 .....
00000200 09 00 0A 00 00 00 04 00 08 00 0B 00 0C 00 01 00 .....
00000210 0D 00 00 00 29 00 01 00 00 00 00 00 09 10 06 B8 ....).....
00000220 00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 06 .....
00000230 00 01 00 00 00 03 00 17 00 00 00 02 00 00 00 01 .....
  
```

图 4.20 字节码中的方法数量

由图 4.20 可知，其值为 4，即 `Test` 类中一共有 4 个方法。可能很多人对此会有疑惑，在 `Test` 源程序中明明只定义了两个方法，为什么字节码文件中却显示有 4 个呢？这是因为在编译期间，编译器会自动为一个类增加 `void <clinit>()` 这样一个方法，其方法名就是“<clinit>”，返回值为 `void`。该方法的作用主要是执行类的初始化，源程序中的所有 `static` 类型的变量都会在这个方法中完成初始化，全部被 `static{}` 所包围的程序都在这个方法中执行（这个后面在讲解 JVM 源码时会详细说明）。同时，在源代码中，并没有为 `Test` 类定义构造函数，因此编译器会自动为该添加一个默认的构造函数（这一点大家应该都知道）。因此，字节码文件会显示 `Test` 类中一共包含 4 个方法。

4.6.2 method_info methods[method_count]

紧跟着 `methods_count` 后面的是 `methods` 结构，这是一个数组，每一个方法的全部细节都包含在里面，包括代码指令。

1. methods 结构组成格式

要分析 `methods` 结构信息，首先需要清楚该结构的数据组成格式，其格式如表 4.7（方法表结构和字段表结构一样）所示。

表 4.7 methods 结构组成

类 型	名 称	数 量
u2	access_flags	1
u2	name_index	1
u2	descriptor_index	1
u2	attributes_count	1
attribute_info	attributes	attributes_count

由表 4.7 可知，方法各个数据项的含义非常相似，仅在访问标志位和属性表集合的可选项上有略微不同。这些字段的含义与上文给出的 fields 结构的字段含义基本相同，因此这里不作具体说明。

其中，JVM 规范为 access_flags 规定了一组可选项值，如表 4.8 所示。

表 4.8 access_flags 可选项值

标 志 名 称	标 志 值	含 义
ACC_PUBLIC	0x0001	字段是否为 public
ACC_PRIVATE	0x0002	字段是否为 private
ACC_PROTECTED	0x0004	字段是否为 protected
ACC_STATIC	0x0008	字段是否为 static
ACC_FINAL	0x0010	字段是否为 final
ACC_SYNCHRONIZED	0x0020	字段是否为 synchronized
ACC_BRIDGE	0x0040	方法是否是由编译器产生的桥接方法
ACC_VARARGS	0x0080	方法是否接受不定参数
ACC_NATIVE	0x0100	字段是否为 native
ACC_ABSTRACT	0x0400	字段是否为 abstract
ACC_STRICTFP	0x0800	字段是否为 strictfp
ACC_SYNTHETIC	0x1000	字段是否为编译器自动产生

由于 ACC_VOLATILE 标志和 ACC_TRANSIENT 标志不能修饰方法，所以 access_flags 中不包含这两项，同时增加 ACC_SYNCHRONIZED 标志、ACC_NATIVE 标志、ACC_STRICTFP 标志和 ACC_ABSTRACT 标志。

2. 第一个方法 void <clinit>()

上面了解了方法描述的信息结构,下面来实际看看 Test.class 字节码文件中的第一个方法究竟是如何描述的。紧跟在 methods_count 后面的就是第一个方法的信息,如图 4.21 所示。

00000190	67 73 01 00 13 5B 4C 6A	61 76 61 2F 6C 61 6E 67	gs...[Ljava/lang
000001a0	2F 53 74 72 69 6E 67 3B	01 00 04 74 65 73 74 0A	/String:...test.
000001b0	00 0F 00 2E 0C 00 2F 66	00 00 00 08 00 00 00 00	...intV
000001c0	61 76 61 00 00 00 00 00	00 03 00 00 0A 5C 6F 75	size...()I...Sou
000001d0	72 63 6C 65 01 00 09 54	65 73 74 6E 6A	rceFile...Test.j
000001e0	61 76 61 00 00 01 00	03 00 00 00 03 00 01 00	ava!.....
000001f0	05 00 06 00 00 00 08 00	07 00 08 00 00 00 00 00
00000200	09 00 0A 00 00 00 04 00	08 00 0B 00 0C 00 01 00
00000210	0D 00 00 00 29 00 01 00	00 00 00 00 09 10 06 B8
00000220	00 0E B3 00 14 B1 00 00	00 02 00 16 00 00 00 06
00000230	00 01 00 00 00 03 00 17	00 00 00 02 00 00 00 01
00000240	00 18 00 0C 00 01 00 0D	00 00 00 46 00 02 00 01F...
00000250	00 00 00 10 2A B7 00 19	2A 06 B5 00 1B 2A 11 1D*...*
00000260	B5 00 1F B1 00 00 00 02	00 16 00 00 00 12 00 04
00000270	00 00 00 01 00 04 00 02	00 09 00 04 00 0F 00 01
00000280	00 17 00 00 00 0C 00 01	00 00 00 10 00 21 00 22!"
00000290	00 00 00 09 00 23 00 24	00 01 00 0D 00 00 00 0E\$....N
000002a0	00 02 00 02 00 00 00 12	B8 00 01 59 B7 00 25 0CY...%L
000002b0	2B 10 08 B8 00 0E B6 00	26 B1 00 00 00 02 00 06	+.....&.....
000002c0	00 00 00 0E 00 03 00 00	00 07 00 08 00 00 00 00
000002d0	00 09 00 17 00 00 00 16	00 02 00 00 00 00 00 00方法属性数量
000002e0	00 2B 00 00 00 00 00 0A	00 2C 00 22 00 01 00 01+....."
000002f0	00 28 00 29 00 01 00 0D	00 00 00 41 00 02 00 02(.....A...
00000300	00 00 00 00 00 00 00 00	00 1B B1 00 00 00 00*+.....
00000310	02 00 16 00 00 00 0A 00	02 00 00 00 0C 00 08 00!.....
00000320	0D 00 17 00 00 00 16 00	02 00 00 00 09 00 21 00
00000330	22 00 00 00 00 00 09 00	05 00 08 00 01 00 01 00"
00000340	31 00 00 00 02 00 32		1.....2

图 4.21 void <clinit>()方法对应的字节码

由图 4.21 可以看出,字节码文件对方法的描述比较复杂,不像前面对魔数、版本号、常量池等信息的描述那么简单。但无论多么复杂的描述,总是遵循其内在的结构逻辑,只要按照 JVM 的规范按图索骥,总是能够分析清楚字节码所要表达的含义。

按照 fields 的结构组成格式,前 2 字节描述 access_flags,即访问标识,由图 4.21 可知其值为 0x0008,对照上文所给出的方法访问标识可选项值的表可知,该值标识该方法的修饰符是 AC_STATIC,也即这是一个 static 类型的静态方法。

接下来的 2 字节描述 name_index,该字段描述的是方法名,其值指向常量池中对应的元素编号。由图 4.21 可知,其值是 0x000B,指向常量池中第 11 号元素,根据上文使用 javap -verbose 命令所打印出的常量池信息可知,常量池中第 11 号元素是<clinit>,即当前所描述的方法名是“<clinit>”,这在上文提到过,该方法是 Java 编译器在编译期间动态添加的类初始化的方法,而非在源程序中定义的。

接下来的 2 字节描述 `descriptor_index`，该字段描述的是方法的入参和出参信息，其值指向常量池中对应的元素编号。由图 4.21 可知，其值是 `0x000C`，指向常量池中第 12 号元素，根据上文打印的常量池表可知，常量池中第 12 号元素是 `()V`，这表示当前方法没有入参（因为是空括号），并且方法的返回值类型是 `void`（`V` 代表 `void`）。这里要注意，按照 JVM 的规范，描述符对入参将严格按照源程序中所定义的参数列表顺序，从左到右依次放入“`()`”内，如方法“`String getAll(int id,String name)`”的描述符为“`(I,Ljava/lang/String;)Ljava/lang/String;`”。

根据这些信息可知，字节码所描述的第一个方法是 `void <clinit>()`。

接下来的 2 字节描述方法所包含的属性的总数量 `attributes_count`，由图 4.21 可知其值为 `0x0001`，表示当前方法一共包含 1 个属性。该字段后面的字节码流将描述详细的属性信息。在分析字节码流中的属性信息之前，有必要先了解 `attributes` 这一字段结构的组成格式。

3. 9 大属性表集合

在 `class` 文件中，属性表、方法表中都可以包含自己的属性表集合，用于描述某些场景的专有信息。本部分内容与 JVM 规范官方文档保持一致，因此这里仅列出大概的概括信息，详细信息可以去阅读官方文档。

与 `class` 文件中其他数据项对长度、顺序、格式的严格要求不同，属性表集合不要求其中包含的属性表具有严格的顺序，并且只要属性的名称不与已有的属性名称重复，任何人实现的编译器都可以向属性表中写入自己定义的属性信息。虚拟机在运行时会忽略不能识别的属性。为了能正确解析 `class` 文件，虚拟机规范中预定义了虚拟机实现必须能够识别的 9 项属性，如表 4.9 所示。

表 4.9 9 大属性

属性名称	使用位置	含 义
Code	方法表	Java 代码编译成的字节码指令
ConstantValue	字段表	final 关键字定义的常量值
Deprecated	类文件、字段表、方法表	被声明为 deprecated 的方法和字段
Exceptions	方法表	方法抛出的异常
InnerClasses	类文件	内部类列表
LineNumberTable	Code 属性	Java 源码的行号与字节码指令的对应关系
LocalVariableTable	Code 属性	方法的局部变量描述
SourceFile	类文件	源文件名称
Synthetic	类文件、方法表、字段表	标识方法或字段是由编译器自动生成的

这 9 种属性中的每一种属性又都是一个复合结构，均有各自的表结构。这 9 种表结构有一个共同的特点，即均由一个 u2 类型的属性名称开始，可以通过这个属性名称来判段属性的类型。该 u2 类型的属性名称指向常量池中对应的元素。

下面描述这 9 种属性具体的复合组成结构。

1) Code 属性

Java 程序方法体中的代码经过 javac 编译器处理后，最终变为字节码指令存储在 Code 属性中。当然不是所有的方法都必须有这个属性（接口中的方法或抽象方法就不存在 Code 属性），Code 属性结构如表 4.10 所示。

表 4.10 Code 属性结构表

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	max_stack	1
u2	max_locals	1
u4	code_length	1
u1	code	code_length
u2	exception_table_length	1
exception_info	exception_table	exception_table_length
u2	attributes_count	1
attribute_info	attributes	attributes_count

Code 属性表中相关字段的含义如下：

- ◎ max_stack，操作数栈深度最大值，在方法执行的任何时刻，操作数栈深度都不会超过这个值。虚拟机运行时根据这个值来分配栈帧的操作数栈深度。
- ◎ max_locals，局部变量表所需存储空间，单位为 Slot。并不是所有局部变量占用的 Slot 之和，当一个局部变量的生命周期结束后，其所占用的 Slot 将分配给其他依然存活的局部变量使用，按此方式计算出方法运行时局部变量表所需的存储空间。
- ◎ code_length 和 code，用来存放 Java 源程序经编译后生成的字节码指令。code_length 代表字节码长度，code 是用于存储字节码指令的一系列字节流。

每一个指令都是一个 u1 类型的单字节，当虚拟机读到 code 中的一个字节码（一个字节能表示 256 种指令，Java 虚拟机规范定义了其中约 200 个编码对应的指令）时，就可以判断出该

字节码代表的指令，指令后面是否带有参数，参数该如何解释，虽然 code_length 占 4 字节，但是 Java 虚拟机规范限制一个方法不能超过 65 535 条字节码指令，如果超过，Javac 将拒绝编译。

2) ConstantValue 属性

ConstantValue 属性通知虚拟机自动为静态变量赋值，只有被 static 关键字修饰的变量（类变量）才可以使用这项属性。其结构如表 4.11 所示。

表 4.11 ConstantValue 属性结构表

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	constantvalue_index	1

可以看出，ConstantValue 属性是一个定长属性，其中 attribute_length 的值固定为 0x00000002，constantvalue_index 为一常量池字面量类型常量索引（class 文件格式的常量类型中只有与基本类型和字符串类型相对应的字面量常量，所以 ConstantValue 属性只支持基本类型和字符串类型）。

对非 static 类型变量（实例变量，如 int a = 123;）的赋值是在实例构造函数<init>中进行的。

对类变量（如 static int a = 123;）的赋值有两种选择，在类构造函数<clinit>方法中或使用 ConstantValue 属性。当前 javac 编译器的选择是，如果变量同时被 static 和 final 修饰（虚拟机规范只要求有 ConstantValue 属性的字段必须设置 ACC_STATIC 标志，对 final 关键字的要求是 javac 编译器自己加入的要求），并且该变量的数据类型为基本类型或字符串类型，就生成 ConstantValue 属性进行初始化；否则在类构造函数<clinit>中进行初始化。

3) Exceptions 属性

该属性列举出方法中可能抛出的受查异常（即方法描述时 throws 关键字后列出的异常），与 Code 属性同级，与 Code 属性包含的异常表不同，其结构如表 4.12 所示。

表 4.12 Exceptions 属性结构表

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	number_of_exceptions	1
u2	exception_index_table	number_of_exceptions

number_of_exceptions 表示可能抛出 number_of_exceptions 种受查异常。

`exception_index_table` 为异常索引集合，一组 `u2` 类型 `exception_index` 的集合，每一个 `exception_index` 为一个指向常量池中一个 `CONSTANT_Class_info` 型常量的索引，代表该受查异常的类型。

4) InnerClasses 属性

该属性用于记录内部类和宿主类之间的关系。如果一个类中定义了内部类，编译器将会为这个类与这个类包含的内部类生成 `InnerClasses` 属性，其结构如表 4.13 所示。

表 4.13 InnerClasses 属性结构表

类 型	名 称	数 量
<code>u2</code>	<code>attribute_name_index</code>	1
<code>u4</code>	<code>attribute_length</code>	1
<code>u2</code>	<code>number_of_classes</code>	1
<code>inner_classes_info</code>	<code>inner_classes</code>	<code>number_of_classes</code>

`inner_classes` 为内部类表集合，一组内部类表类型数据的集合，`number_of_classes` 即为集合中内部类表类型数据的个数。

每一个内部类的信息都由一个 `inner_classes_info` 表来描述，`inner_classes_info` 表结构如表 4.14 所示。

表 4.14 inner_classes_info 表结构

类 型	名 称	数 量
<code>u2</code>	<code>inner_class_info_index</code>	1
<code>u2</code>	<code>outer_class_info_index</code>	1
<code>u2</code>	<code>inner_name_index</code>	1
<code>u2</code>	<code>inner_name_access_flags</code>	1

`inner_class_info_index` 和 `outer_class_info_index` 指向常量池中 `CONSTANT_Class_info` 类型常量索引，该 `CONSTANT_Class_info` 类型常量指向常量池中 `CONSTANT_Utf8_info` 类型常量，分别为内部类的全限定名和宿主类的全限定名。

`inner_name_index` 指向常量池中 `CONSTANT_Utf8_info` 类型常量的索引，为内部类名称，如果为匿名内部类，则该值为 0。

`inner_name_access_flags` 类似于 `access_flags`，是内部类的访问标志，该标识的可选项值与前面描述类的访问属性的可选项值一致。

5) LineNumberTable 属性

用于描述 Java 源码的行号与字节码行号之间的对应关系，非运行时必需属性，会默认生成至 class 文件中，可以使用 javac 的 -g:none 或 -g:lines 命令关闭或要求生成该项属性信息，其结构如表 4.15 所示。

表 4.15 LineNumberTable 属性结构表

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	line_number_table_length	1
line_number_info	line_number_table	line_number_table_length

line_number_table 是一组 line_number_info 类型数据的集合，其所包含的 line_number_info 类型数据的数量为 line_number_table_length。line_number_info 结构如表 4.16 所示。

表 4.16 Line_number_info 属性结构表

类 型	名 称	数 量	说 明
u2	start_pc	1	字节码行号
u2	line_number	1	Java 源码行号

不生成该属性的最大影响是：

- ◎ 抛出异常时，堆栈将不会显示出错的行号。
- ◎ 调试程序时无法按照源码设置断点。

6) LocalVariableTable 属性

用于描述栈帧中局部变量表中的变量与 Java 源码中定义的变量之间的关系，非运行时必需属性，默认不会生成至 class 文件中，可以使用 javac 的 -g:none 或 -g:vars 命令关闭或要求生成该项属性信息，其结构如表 4.17 所示。

表 4.17 LocalVariableTable 属性结构表

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	local_variable_table_length	1
local_variable_info	local_variable_table	local_variable_table_length

`local_variable_table` 是一组 `local_variable_info` 类型数据的集合,其所包含的 `local_variable_info` 类型数据的数量为 `local_variable_table_length`。`local_variable_info` 结构如表 4.18 所示。

表 4.18 `local_variable_info` 结构表

类 型	名 称	数 量	说 明
u2	start_pc	1	局部变量的生命周期开始的字节码偏移量
u2	length	1	局部变量作用范围覆盖的长度
u2	name_index	1	指向常量池中 <code>CONSTANT_Utf8_info</code> 类型常量的索引, 局部变量名称
u2	descriptor_index	1	指向常量池中 <code>CONSTANT_Utf8_info</code> 类型常量的索引, 局部变量描述符
u2	index	1	局部变量在栈帧局部变量表中 Slot 的位置, 如果这个变量的数据类型为 64 位类型 (long 或 double), 它占用的 Slot 为 index 和 index+1 这 2 个位置

`start_pc + length` 即为该局部变量在字节码中的作用域范围。

不生成该属性的最大影响是:

- ◎ 当其他人引用这个方法时,所有的参数名称都将丢失,IDE 可能会使用诸如 `arg0`、`arg1` 之类的占位符代替原有的参数名称,对代码运行无影响,会给代码的编写带来不便。
- ◎ 调试时调试器无法根据参数名称从运行上下文中获取参数值。

7) `SourceFile` 属性

用于记录生成这个 class 文件的源码文件名称,为可选项,可以使用 `javac` 的 `-g:none` 或 `-g:source` 命令关闭或要求生成该项属性信息,其结构如表 4.19 所示。

表 4.19 `SourceFile` 属性结构表

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1
u2	sourcefile_index	1

可以看出, `SourceFile` 属性是一个定长属性, `sourcefile_index` 是指向常量池中一个 `CONSTANT_Utf8_info` 类型常量的索引,常量的值为源码文件的文件名。

对大多数文件,类名和文件名是一致的,少数特殊类除外(如内部类),此时如果不生成这项属性,当抛出异常时,堆栈中将不会显示出错误代码所属的文件名。

8) `Deprecated` 属性和 `Synthetic` 属性

这两个属性都属于标志类型的布尔属性，只存在有和没有的区别，没有属性值的概念。

Deprecated 属性表示某个类、字段或方法已经被程序作者定为不再推荐使用，可在代码中使用@Deprecated 注解进行设置。

Synthetic 属性表示该字段或方法不是由 Java 源码直接产生的，而是由编译器自行添加的(当然也可设置访问标志 ACC_SYNTHETIC，所有由非用户代码产生的类、方法和字段都应当至少设置 Synthetic 属性和 ACC_SYNTHETIC 标志位中的一项，唯一的例外是实例构造函数<init>和类构造函数<clinit>)。

这两项属性的结构（当然 attribute_length 的值必须为 0x00000000）如表 4.20 所示。

表 4.20 属性结构

类 型	名 称	数 量
u2	attribute_name_index	1
u4	attribute_length	1

4. 继续第一个方法

上面对 JVM 所定义的 9 大属性表进行了简介，我们有了对属性表的基本结构认识，接下来便可以继续分析上文中第一个方法的字节码。

Test.class 字节码文件中，第 0x020D 和第 0x020E 两个位置的字节一起组成了第一个方法的属性表数量 attributes_count，紧跟在属性表数量后面的是 attributes 数组的第一个属性。上文提到，虽然 JVM 所支持的 9 大属性，其相互之间格式相差甚远，但是都会以一个 u2 类型的属性名称开始，JVM 根据名称便可知道当前描述的到底是这 9 大属性中的哪一个属性。Test.class 字节码文件中，紧跟在 attributes_count 后面的是第 0x020F 与第 0x0210 这两个字节，其值是 0x000D，如图 4.22 所示。



图 4.22 void <clinit>()方法的属性名称索引

属性表的名称索引指向常量池中对应的位置，根据上文所打印的 Test.class 常量池信息可知，常量池中第 0x000D（即 13）号元素的值是 Code，正是上文所介绍的 9 大属性中的 Code 方法表。根据上文所介绍的 Code 方法表属性的组成结构可知，紧跟在 u2 类型的属性名称后面的是 u4

类型的 `attribute_length`，对照图 4.22 可知，`attribute_length` 值为 `0x00000029`（如图 4.23 所示），对应十进制的 41。

000001f0	05 00 06 00 00 00 08 00	07 00 08 00 00 00 00 00
00000200	09 00 0A 00 00 00 04 00	08 00 0B 00 0C 00 01 00
00000210	0D 00 00 00 29 00 01 00	00 00 00 00 09 10 06 B8
00000220	00 0E B3 00 14 B1 00 00	00 02 00 16 00 00 00 06
00000230	00 01 00 00 00 03 00 17	00 00 00 02 00 00 00 01
00000240	00 18 00 0C 00 01 00 0D	00 00 00 46 00 02 00 01F....

图 4.23 `void <clinit>()` 方法的属性长度

Code 属性组成结构中紧跟在 `attribute_length` 后面的是 `u2` 类型的 `max_stack` 和 `u2` 类型的 `max_locals`，其值分别是 1 和 0，如图 4.24 所示。

000001f0	05 00 06 00 00 00 08 00	07 00 08 00 00 00 00 00
00000200	09 00 0A 00 00 00 04 00	08 00 0B 00 0C 00 01 00
00000210	0D 00 00 00 29 00 01 00	00 00 00 00 09 10 06 B8
00000220	00 0E B3 00 14 B1 00 00	00 02 00 16 00 00 00 06
00000230	00 01 00 00 00 03 00 17	00 00 00 02 00 00 00 01
00000240	00 18 00 0C 00 01 00 0D	00 00 00 46 00 02 00 01F....

图 4.24 `void <clinit>()` 方法的最大栈深和局部变量表

紧跟在 `max_locals` 后面的是 `u4` 类型的 `code_length`，其值是 9，如图 4.25 所示。

000001f0	05 00 06 00 00 00 08 00	07 00 08 00 00 00 00 00
00000200	09 00 0A 00 00 00 04 00	08 00 0B 00 0C 00 01 00
00000210	0D 00 00 00 29 00 01 00	00 00 00 00 09 10 06 B8
00000220	00 0E B3 00 14 B1 00 00	00 02 00 16 00 00 00 06
00000230	00 01 00 00 00 03 00 17	00 00 00 02 00 00 00 01
00000240	00 18 00 0C 00 01 00 0D	00 00 00 46 00 02 00 01F....

图 4.25 `void <clinit>()` 方法的字节码指令长度

`code_length` 之后的字节码流是 `code` 属性，`code` 属性开始真正描述 Java 方法所对应的字节码指令，这是 Java 的精华所在。字节码指令所占的字节码长度由 `code_length` 决定，由于 `code_length` 的值是 9，因此 `code_length` 后面的 9 字节都用于描述字节码指令。这 9 个字节码的值是：10 06 B8 00 0E B3 00 14 B1，如图 4.26 所示。

000001f0	05 00 06 00 00 00 08 00	07 00 08 00 00 00 00 00
00000200	09 00 0A 00 00 00 04 00	08 00 0B 00 0C 00 01 00
00000210	10 00 00 00 29 00 01 00	00 00 00 00 09 10 06 B8
00000220	00 0E B3 00 14 B1 00 00	00 02 00 16 00 00 00 06
00000230	00 01 00 00 00 03 00 17	00 00 00 02 00 00 00 01
00000240	00 18 00 0C 00 01 00 0D	00 00 00 46 00 02 00 01F....

图 4.26 `void <clinit>()` 方法的字节码指令

JVM 是基于栈的指令集系统，其设计的指令仅占 1 字节，由于 1 字节最多只能描述 256 种指令，所以 JVM 的指令总数只有 200 多个。同时，JVM 的指令属于一元操作数类型，其后面只有一个操作数（当然，很多指令后不跟操作数，例如 `return`）。正因如此，对于 JVM 指令需要区别对待，有些字节是代表指令，但是有些字节则代表数据（专业术语叫作“操作数”），而不是指令。

下面来分析当前方法的指令逻辑。从第一字节开始，第一字节一定是代表指令，而不能是数字（否则连 JVM 自己都不知道真正的指令到底从哪里开始）。第一字节是 `0x10`，查询 JVM 的指令集可知，其含义如下：

指令码	助记符	说 明
0x10	bipush	将 int、float 或 String 型常量值从常量池中推送至栈顶

该指令后面会跟 1 字节作为操作数，因此需要连着两字节一起看。其后面的 1 字节是 `0x06`，因此第一个指令所表述的含义是：将整型数字 6 推送至栈顶。如图 4.27 所示。

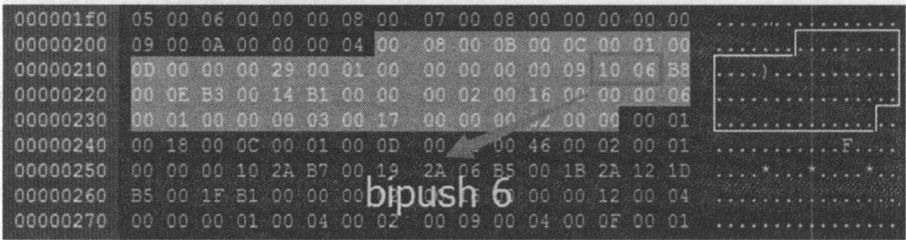


图 4.27 `void <clinit>()` 方法的第 1 条字节码指令

第一个指令的字节码流结束后，紧跟其后的第一字节一定还是指令而非数值。跟在 `0x1006` 后面的第一字节是 `0xB8`，查询指令集可知，其含义如下：

指令码	助记符	说 明
0xB8	invokestatic	调用静态方法

该指令属于一元指令码，其后面会跟一个占 2 字节的数据来指向常量池中的方法编号，其后面的 2 字节是 `0x000E`，指向常量池中第 14 号元素。根据上文所打印的常量池信息可知，常量池中第 14 号元素信息如下：

```
const #14 = Method #15.#17; // java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
const #15 = class #16; // java/lang/Integer
const #16 = Asciz java/lang/Integer;
const #17 = NameAndType #18:#19; // valueOf:(I)Ljava/lang/Integer;
const #18 = Asciz valueOf;
const #19 = Asciz (I)Ljava/lang/Integer;;
```


第 14 号常量池元素类型是 Method，其引用了第 15 和第 17 号常量池元素，第 17 号元素又引用了第 18 和第 19 号元素，将它们最终拼凑起来，该方法是：

```
java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
```

该方法信息分别描述了方法所在的类、方法名、入参和返回值。

第二条指令一共占 3 字节，其所在位置如图 4.28 所示。



图 4.28 void <clinit>()方法的第 2 条字节码指令

分析完第二条指令，接下来开始分析第三条指令。同理，紧跟在第二条指令后面的第一字节一定是一个 JVM 指令，而非数据。该字节的值是 0xB3，查询 JVM 指令集可知，其含义如下：

指令码	助记符	说 明
0xB3	putstatic	用栈顶的值为指定的类的静态域赋值

该指令需要跟一个表明域编号的参数（占 2 字节），因此该指令最终占 3 字节，如图 4.29 所示。



图 4.29 void <clinit>()方法的第 3 条字节码指令

后面 2 字节的值是 0x0014，对应十进制数 20，表明其指向常量池中第 20 号元素。常量池中第 20 号元素的信息如下：

```
const #1 = class          #2; // Test
const #7 = Asciz          si;
const #8 = Asciz          Ljava/lang/Integer;;
const #20 = Field          #1.#21; // Test.si:Ljava/lang/Integer;
const #21 = NameAndType   #7:#8; // si:Ljava/lang/Integer;
```

第 20 号元素引用了常量池中其他元素，最终拼凑起来的含义是：

`Test.si:Ljava/lang/Integer`

这表明最终引用的是 `Test` 类中的 `si` 静态变量。于是第三条指令的含义就很清晰了，为 `Test` 类中的 `si` 静态变量赋值（值来自栈顶）。

第三条指令分析完了，接着分析后续指令。第三条指令后面紧接着的第一个字节码是 `0xB1`，查询 JVM 指令集可知，其代表的含义如下：

指令码	助记符	说明
<code>0xB1</code>	<code>return</code>	从当前方法返回 <code>void</code>

由于返回的是 `void`，因此该指令后面没有操作符。

前面提到，当前方法（即 `void <clinit>()`）的 `code_length` 是 9，即该方法对应的指令一共占 9 字节，而上面所分析的 4 个指令正好占了 9 字节，因此第一个方法的 `Code` 属性到此结束。

刚才是逐条分析各个指令的含义，但仅关注单个指令并不能清楚地知道程序想要干什么，因此需要将一个 Java 方法所对应的全部指令连起来一起看。上文使用 `javap -verbose` 命令分析 `class` 文件时，会将每一个 Java 方法所对应的 JVM 字节码打印出来，如下：

```
static {};  
Code:  
Stack=1, Locals=0, Args_size=0  
0:  bipush 6  
2:  invokestatic  #14; //Method  
java/lang/Integer.valueOf:(I)Ljava/lang/Integer;  
5:  putstatic  #20; //Field si:Ljava/lang/Integer;  
8:  return
```

这段指令的含义是：将数据 6 推送至栈顶，然后调用 `java.lang.Integer.valueOf(int i)` 方法将其转换为 `java.lang.Integer` 类型，最后再将转换后的结果赋值给 `Test.si` 这个静态变量。由于 `Test` 类中只有一个静态变量 `si`，因此在整个类初始化阶段的确只需要将 `si` 进行初始化。

5. 第一个方法中的属性表

JVM 一共支持 9 大属性，`Code` 就是其中一个属性。然而 `Code` 属性中也会引用其他 8 类属性。`Code` 属性结构的最后 2 个组成部分分别是 `attributes_count` 和 `attributes`，分别代表所引用的属性总数和属性信息。其中，`attributes_count` 占 4 字节，`attributes` 所占字节数需要视具体情况而定。

在 `Test.class` 字节码文件中，描述完第一个方法（`void <clinit>()`）的 `Code` 属性后，接下来的字节流便开始描述该方法所引用的其他属性信息。开始的 4 字节表示 `attributes_count`，其内

容如图 4.30 所示。

000001f0	05 00 06 00 00 00 08 00	07 00 08 00 00 00 00 00
00000200	09 00 0A 00 00 00 04 00	08 00 0B 00 0C 00 01 00
00000210	0D 00 00 00 29 00 01 00	00 00 00 00 09 10 06 B8)
00000220	00 0E B3 00 14 B1 00 00	00 02 00 16 00 00 00 06
00000230	00 01 00 00 00 03 00 17	00 00 00 02 00 00 00 01
00000240	00 18 00 0C 00 01 00 0D	00 00 00 46 00 02 00 01F...
00000250	00 00 00 10 2A B7 00 19	2A 06 B5 00 1B 2A 12 1D*.....*
00000260	B5 00 1F B1 00 00 00 02	00 16 00 00 00 12 00 04

图 4.30 void <clinit>()方法的属性长度

由图 4.30 可知, attributes_count 的值是 2, 表示 void <clinit>()方法中一共引用了两个其他的属性表。

上文提到,JVM 的 9 大属性表虽然结构各不相同,但是都以 u2 类型、占 2 字节的 tag 开头,JVM 通过 tag 来区分当前究竟是哪一种属性。由图 4.30 可知,紧跟在 attributes_count 后面的 2 字节的值是 0x0016,如图 4.31 所示

000001f0	05 00 06 00 00 00 08 00	07 00 08 00 00 00 00 00
00000200	09 00 0A 00 00 00 04 00	08 00 0B 00 0C 00 01 00
00000210	0D 00 00 00 29 00 01 00	00 00 00 00 09 10 06 B8)
00000220	00 0E B3 00 14 B1 00 00	00 02 00 16 00 00 00 06
00000230	00 01 00 00 00 03 00 17	00 00 00 02 00 00 00 01
00000240	00 18 00 0C 00 01 00 0D	00 00 00 46 00 02 00 01F...
00000250	00 00 00 10 2A B7 00 19	2A 06 B5 00 1B 2A 12 1D*.....*
00000260	B5 00 1F B1 00 00 00 02	00 16 00 00 00 12 00 04

图 4.31 void <clinit>()方法属性 tag

由图 4.31 可知,当前属性的 tag 值对应十进制的 22,根据上文所打印的常量池信息可知,常量池中第 22 号元素信息如下:

```
const #22 = Asciz      LineNumberTable;
```

由此可知,当前所描述的属性是 LineNumberTable,该属性用于记录源代码与字节码指令之间的行号对应关系。根据上文对 9 大属性的简介可知,描述 LineNumberTable 属性的字节码流结构包括: u2 类型的 attribute_name_index、u4 类型的 attribute_length、u2 类型的 line_number_table_length 和 line_number_table 行号对应关系表。其中, u2 类型的 attribute_name_index 刚才已经分析过,即指向常量池中第 22 号元素。紧跟在 attribute_name_index 后面的是 u4 类型的 attribute_length,其值如图 4.32 所示。

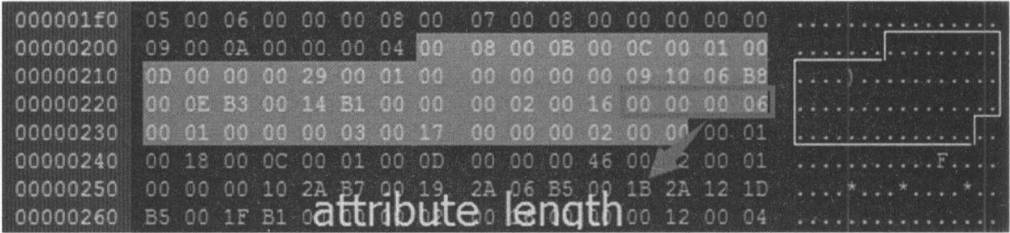


图 4.32 void <clinit>()方法行号表属性元素长度

由图 4.32 可知，其值是 6，表明 LineNumberTable 这个属性接下来还占 6 字节，6 字节之后的字节码流就不属于当前属性了。

紧跟在 attribute_length 结构后面的是 u2 类型的 line_number_table_length，占 2 字节。该结构标记其后面的 line_number_table 的元素数量，其内容如图 4.33 所示。



图 4.33 void <clinit>()方法行号表的第一个元素的 tag

由图 4.33 可知，其值是 1，表明其后面的 line_number_table 的元素数量是 1。由于 attribute_length 的值是 6，而 line_number_table_length 的值占去了 2 字节，因此整个 LineNumberTable 属性还剩下最后 4 字节，最后 4 字节的内容如图 4.34 所示。



图 4.34 void <clinit>()方法行号表的字节码指令偏移与源码行号的对应关系

这最后 4 字节皆用于描述 line_number_table 行号对应关系表结构，line_number_table 是一组 line_number_info 类型数据的集合，即该表结构可以认为是一个标准的数组结构，其中所有元素类型都是 line_number_info 结构，而该结构由 2 个子结构构成：line_number_table_length

和 line_number_info，如表 4.21 所示：

表 4.21 子结构

类 型	名 称	数 量	说 明
u2	start_pc	1	字节码行号（也即字节码指令偏移位置）
u2	line_number	1	Java 源码行号

每一个子结构各占 2 字节，因此 line_number_table 结构所占的字节数一定是 4 的倍数。据此可以推测，由于留给 line_number_table 结构的只剩下 4 字节，因此这个结构数组中仅包含一个 line_number_info 元素。由图 4.34 可知，其值是 0x00000003，前两字节代表字节码指令偏移位置，值是 0，后两字节代表 Java 源码行号，值是 3。

上文使用 javap -verbose 命令所打印的常量池表中，包含 LineNumberTable 属性的描述，如图 4.35 所示。

```
static {}:  
Code:  
  $stack=1, Locals=0, Args_size=0  
  0:  bipush 6  
  2:  invokestatic    #14; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;  
  5:  putstatic        #20; //Field si:Ljava/lang/Integer;  
  8:  return  
LineNumberTable:  
  line 3: 0
```

图 4.35 使用 javap -verbose 命令所打印出的 void <clinit>()方法行号表

上面完成了 Test.class 字节码文件中第一个方法 void <clinit>()中所引用的第一个属性 LineNumberTable 的分析，由于该方法一共引用了 2 个属性，因此继续分析第二个属性。同理，第二个属性以一个 u2 类型的 tag 开头，如图 4.36 所示。

```
000001f0 05 00 06 00 00 00 08 00 07 00 08 00 00 00 00 00 .....  
00000200 09 00 0A 00 00 00 04 00 08 00 0B 00 0C 00 01 00 .....  
00000210 0D 00 00 00 29 00 01 00 00 00 00 09 10 06 B8 .....  
00000220 00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 06 .....  
00000230 00 01 00 00 00 03 00 17 00 00 00 02 00 00 00 01 .....  
00000240 00 18 00 0C 00 01 00 0D 00 00 00 46 00 02 00 01 .....F....  
00000250 00 00 00 10 2A B7 00 19 2A 06 B5 00 1B 2A 12 1D ....*...*...*..  
00000260 B5 00 1F B1 00 00 00 02 00 16 00 00 00 12 00 04 .....
```

图 4.36 void <clinit>()方法的第二个属性 tag

其值是 0x0017，对应十进制的 23，表明该属性的名称指向常量池中第 23 号元素，常量池中该元素信息如下：

```
const #23 = Asciz      LocalVariableTable;
```

由此可知，当前属性类型是 LocalVariableTable，该属性描述栈帧中局部变量表中的变量与 Java 源码中定义的变量之间的关系。根据上文对该属性简介可知，该属性组成结构依次是：u2 类型的 attribute_name_index、u4 类型的 attribute_length、u2 类型的 local_variable_table_length 和最后的 local_variable_table。紧跟在 attribute_name_index 后面的结构是 u4 类型的 attribute_length，其值如图 4.37 所示。



图 4.37 void <clinit>()方法的第二个属性长度

由图 4.37 可知，其值是 2，表明其后面的信息一共只占 2 字节。紧跟在 attribute_length 结构后面的结构是 u2 类型的 local_variable_table_length，由图 4.37 可知，其值是 0x0000，这表明当前方法并没有引用任何类成员变量。由于 local_variable_table_length 值是 0，因此字节码文件中就不再为其后面的结构 local_variable_table 分配空间。至此 Test.class 字节码文件对 LocalVariableTable 属性的描述结束。

现在，对 Test.class 字节码文件中第一个方法的分析全部完成。字节码文件中最复杂的部分应该就是对 Java 方法的描述，虽然 Test 类的 <clinit>() 方法很简单，但是相信各位道友应该有能力通过对这一简单方法的分析，窥一斑而知全豹，举一反三，独立分析其他更加复杂的 Java 方法的字节码信息。当然，在日常开发工作中，很少需要为了解决问题而去分析字节码文件，通过对字节码文件的分析，更主要的还是为了后面对 JVM 执行引擎工作原理的分析做铺垫，毕竟字节码文件是格式化的，JVM 也是基于同样的规范去解读字节码文件的，如果对字节码文件格式缺乏了解，那么在阅读 JVM 源码时将不太容易跟得上 JVM 作者们的思路和节奏，给源码阅读带来不少障碍。

6. <clinit>()方法的字节码长度

上文在分析 void <clinit>() 方法时，描述该方法的 Code 属性中有一个结构专门描述该方法

所占字节数，该结构是 `code_length`。通过上面的分析可知，`<clinit>()` 方法的 `code_length` 值是 41，这个值代表的范围是多大呢？请看图 4.38 所示。



图 4.38 `<clinit>()` 方法的 `Code` 属性的 `code_length` 结构所标识的范围

如图 4.38 所示，`code_length` 结构占 4 字节，从字节码文件中的第 0x0211 至第 0x0214，其后面的 41 个字节码流全部用于描述方法本身的指令、所引用的其他属性信息，正好到字节码文件中的第 0x023D 位置结束，过了这个位置，后面的字节码流信息就不再属于当前方法的范畴，由此可知，`code_length` 的作用域范围是当前方法的后续所有属性的长度总和。

`Test.class` 一共包含 4 个方法，下文不会再一个一个详细分析它们的字节码信息，而是会以 `code_length` 为指向标，标识出剩余 3 个方法的界限，具体的内容请读者自行完成。

7. 第二个方法

上文提到，`methods` 结构的组成如表 4.22 所示。

表 4.22 `methods` 结构组成

类 型	名 称	数 量
u2	access_flags	1
u2	name_index	1
u2	descriptor_index	1
u2	attributes_count	1
attribute_info	attributes	attributes_count

第二个方法的 `access_flags`、`name_index`、`descriptor_index` 和 `attributes_count` 的信息如图 4.39 所示。

00000210	0D 00 00 00 29 00 01 00 00 00 00 00 09 10 06 B8)
00000220	00 0E B3 00 14 B1 00 00 00 02 00 16 00 00 00 06
00000230	00 01 00 00 00 03 00 17 00 00 00 02 00 00 00 01
00000240	00 18 00 0C 00 01 00 0D 00 00 00 46 00 02 00 01F..
00000250	01 00 00 10 2A 37 00 19 2A 06 B5 00 1B 2A 12 7*.*.*.*
00000260	B1 00 1F 11 00 00 02 00 16 00 00 00 12 00 04
00000270	00 00 00 0 00 01 00 02 00 09 00 04 00 0F 00 01access_flag
00000280	00 17 00 0 00 00 00 01 00 00 00 10 00 21 00 22
00000290	00 00 00 09 00 23 00 24 00 01 00 0D 00 00 00 4E#. \$.N
000002a0	00 00 00 02 00 00 00 00 00 00 00 00 00 00 00Y..%L
000002b0	2B 10 08 B8 00 0E B6 00 20 B1 00 00 00 02 00 16	+.....&.....
000002c0	00 00 00 00 00 00 00 00 00 07 00 08 00 08 00 11
000002d0	00 09 00 17 00 00 00 16 00 02 00 00 00 12 00 2A*

图 4.39 第二个方法的字节码信息

其值分别是 0x01、0x18、0x0C 和 0x01，对照上文所给出的相关选项值信息可知，当前方法的访问标识是 public，当前方法的名字和入参与返回值分别指向常量池中第 24 号、第 12 号元素，信息如下：

```
const #12 = Asciz      ()V;
const #24 = Asciz      <init>;
```

根据该信息可知，当前方法描述的是类的构造函数，其返回值类型是 void，无入参，因此是默认构造函数。注意，字节码文件中名字为<init>的方法表示的是类的构造函数，上文所描述的第一个方法名是<clinit>，这是类型的初始化方法。这两者的区别是：当 JVM 决定加载某个类型时，会调用<clinit>()方法，而当 JVM 决定实例化某个类型时，会调用<init>方法。一个是类型初始化，一个是类实例的初始化，两者有本质上的区别。并且<clinit>()一定先于<init>()方法被调用。

当前方法的 attribute_count 值是 1，表示当前方法引用了一个属性。

8. 第三个方法

第三个方法的 access_flags、name_index、descriptor_index 和 attributes_count 的信息如图 4.40 所示。

该方法的 access_flag 值是 9，对照上文所给出的方法的 access_flags 可选值可知，该值是由表示 ACC_PUBLIC 的 0x0001 和表示 ACC_STATIC 的 0x0008 合成的，因此当前方法的访问修饰符为 public static。

00000280	00 17 00 00 00 0C 00 01	00 00 00 10 00 21 00 22!."
00000290	00 00 00 09 00 23 00 24	00 01 00 0D 00 00 00 4E	...#.\$.....N
000002a0	00 02 00 02 00 00 00 12	BB 0C 01 59 B7 00 25 4CY..\$L
000002b0	2B 10 03 B8 03 0E B6 07	26 B1 07 00 00 02 00 16	+.....&.....
000002c0	00 00 00 0E 00 03 00 00	00 07 00 08 00 08 00 11
000002d0	00 09 00 17 00 00 00 16	00 00 00 00 00 00 00 00*
000002e0	00 2B 00 00 00 08 00 0A	00 2C 00 22 00 01 00 01	..+....."
000002f0	00 2F 00 29 00 01 00 0D	00 00 00 41 00 02 00 02	..(.).....A..
00000300	00 00 00 00 00 00 00 00	2D B3 00 1B B1 00 00 00	...*+..-.....
00000310	02 00 16 00 00 00 0A 00	02 00 00 00 0C 00 08 00
00000320	00 00 17 00 00 00 16 00	00 00 00 00 00 00 00 00!
00000330	00 00 00 00 00 00 00 00	05 00 08 00 01 00 01 00"
00000340	31 00 00 00 02 00 32		1.....2

图 4.40 第三个方法的字节码信息

同时，该方法的 `name_index` 值为 `0x23`，找到常量池中索引号为 35 的元素，其值如下：

```
const #35 = Asciz      main;
```

由此可知，第三个方法描述的是 `main()` 主函数。关于该方法的具体信息，相信各位道友在看完上面对第一个方法的详细分析后，完全有能力自行研究，这里限于篇幅便不再逐个详细解析了。

除了本章所分析的魔数、版本号、常量池、字段、方法等属性，Java class 字节码文件中还有很多其他属性，这里也不逐个深入分析了。通过对上面这几个属性的分析，认真的道友一定能够把握住字节码格式的套路。在分析属性时，只需要按照 JVM 官方规范所指定的格式，肯定能够分析出字节码文件中所隐藏的信息。

4.7 本章回顾

本章以一个具体的 Java 程序为例，分析了其对应的字节码文件中的数据。可以发现，Java 编译器在生成 Java class 字节码文件时，全是按套路出牌的，那么我们在分析时也按照套路走，便不难理解字节码文件的内容。

字节码文件中最重要的是 Java 方法所对应的字节码指令（至少笔者这么认为），Java 源程序的逻辑都封装在字节码指令中。对字节码指令在字节码文件中的存储方式有了透彻的理解，便意味着你对 JVM 执行引擎入门了^_^。

第 5 章

常量池解析

本章摘要

- ◎ Java 字节码常量池的内存分配链路
- ◎ oop-klass 模型
- ◎ 常量池的解析原理

前文讲述了 JVM 执行引擎的核心机制，以及 Java class 字节码文件的格式。从本章开始，继续剖析 JVM 内部的源码。JVM 要完成 Java 逻辑的执行，必须能够“读懂”Java 字节码文件。而 Java 字节码文件从总体上而言，其实主要包含三部分：常量池、字段信息和方法信息。其中常量池存储了字段和方法的相关符号信息，因此对常量池的解读便成为解读 Java 字节码文件的基础。本章主要分析 JVM 解析 Java 字节码文件中常量池的逻辑。本章内容难度属于中等，只要你稍微具备一点 C/C++ 基础便能读懂，当然，即使完全不具备 C/C++ 基础，读懂本章也不是难事，大家都是写程序的，而写程序的小伙伴们，智商都是蛮高的^_^。

常量池是 Java 字节码文件的核心，因此也是 JVM 解析字节码的重头戏。要注意的是，这里所说的常量池并不等同于 JVM 内存模型中的常量区，这里的常量池仅仅是文件中的一堆字节码而已。但是字节码文件中的常量池与 JVM 内存区的常量区之间却有着千丝万缕的联系，因为 JVM 最终会将字节码文件中的常量池信息进行解析，并存储到 JVM 内存模型中的常量区。字节码文件中的常量池是 Java 编译器对 Java 源代码进行语法解析后的产物，只是这种初步解析产生的结果比较粗糙，里面包含了各种引用，信息不够直观。而 JVM 根据字节码文件中的常量池信息再进行二次解析，这种解析目标清晰，直奔终点，会还原出所有常量元素的全部信息，让内存与你所编写的 Java 源代码保持一致。

在字节码文件中，用于描述常量池结构的字节码流所在的块区紧跟在魔数和版本号之后，因此 JVM 在解析完魔数与版本号后，接着便开始解析常量池。JVM 解析 Java 类字节码文件的接口是 `ClassFileParser::parseClassFile()`，该接口内部解析的总体步骤如图 5.1 所示。

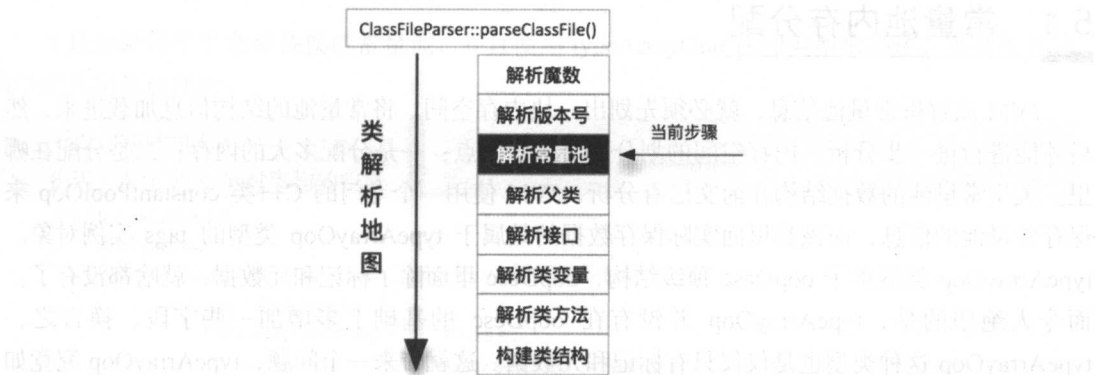


图 5.1 JVM 解析 Java 字节码文件的总体步骤

本章主要讲解 JVM 对 Java 字节码文件中常量池信息的解析。本章以及后续几个章节会详细讲解 JVM 解析常量池、类变量和类方法的过程，为了使思路上保持连贯性，后续章节也会贴出该示意图。

JVM 中对字节码常量池信息的解析的主要链路如图 5.2 所示。

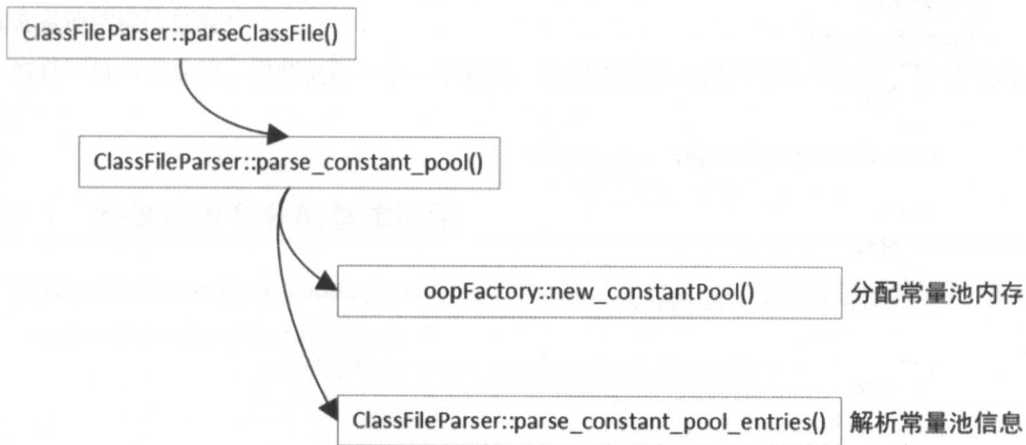


图 5.2 解析常量池的主要链路

由图 5.2 可以看出，JVM 对常量池的解析主要分为两步：第一步是为常量池分配内存，第二步是解析常量池信息。下面就从这两个部分分析常量池解析的过程。

5.1 常量池内存分配

JVM 欲解析常量池信息，就必须先划出一块内存空间，将常量池的结构信息加载进来，然后才能进行进一步分析。内存空间的划分主要考虑两点：一是分配多大的内存；二是分配在哪里。关于常量池的数据结构在前文已有分析，JVM 使用一个专门的 C++ 类 `constantPoolOop` 来保存常量池的信息，而该类里面实际保存数据的是属于 `typeArrayOop` 类型的 `_tags` 实例对象。`typeArrayOop` 类继承于 `oopDesc` 顶级结构，`oopDesc` 里面除了标记和元数据，就啥都没有了。而令人绝望的是，`typeArrayOop` 并没有在 `oopDesc` 的基础上多增加一些字段，换言之，`typeArrayOop` 这种类型也是仅仅只有标记和元数据。这就带来一个问题，`typeArrayOop` 究竟如何对复杂的常量池信息进行描述的呢？其所带来的关联问题就是，JVM 怎样根据 `typeArrayOop` 去分配到合适大小的内存呢？

对于天生就适合领域建模的 Java 语言而言，如果要描述某类对象，一般会为其建立对应的数据结构模型。拿大伙儿喜闻乐见的学生为例，如果使用 Java 来建模，则至少会是这个样子：

清单：Student.Java

作用：演示 Java 类对事物的定义

```
// 学生类型
class Student{
    /**
     * 年龄
     */
    private Integer age;

    /**
     * 身高
     */
    private Integer height;

    /**
     * 名字
     */
    private String name;

    /**
     * 年级
     */
}
```



```
private Integer grade;
}
```

可以看出, Java 语言所定义的这种类型能够对“学生”这种“事物”进行精确描述,同时,数据结构一旦定义好,则意味着其所占用的内存空间也已经确定。

可是如果将学生类型替换成常量池,并且使用 `typeArrayOop` 这种类型来描述,则其所描述的学生类型是这样的:

清单: `Student.Java`

作用: 演示 Java 类对事物的定义

```
class oopDesc {
    private:
        volatile markOop _mark; //线程锁等标记
        union _metadata {
            wideKlassOop _klass;
            narrowOop _compressed_klass;
        } _metadata; //元数据, 自引用
}
```

很显然, 直接根据 `typeArrayOop` 这种类型, 根本就看不出其所描述的对象究竟包含哪些属性, 更无法据此计算出其所描述的事物应该占用多大内存。看来这个类真的如同其名字一样, 只描述数组信息。而事实上, 常量池也的确类似于数组, 不同的 Java 类所生成的常量池信息是不同的, 因此也无法使用一种预先定义好的数据模型去描述。但是, 常量池又不是严格意义上的数组, 因为其每种元素成员的类型并不相同。那么 `typeArrayOop` 如何描述常量池所需内存空间和常量池结构信息呢?

饭是一口一口吃的, 问题也得一个一个解决。我们先将目光回归到本节主题: 常量池内存分配。

5.1.1 常量池内存分配总体链路

在 `ClassFileParser::parse_constant_pool()` 函数中, 通过下面这行代码实现常量池内存分配:

```
onstantPoolOop constant_pool =
    oopFactory::new_constantPool(length,
                                  oopDesc::IsSafeConc,
                                  CHECK_(nullHandle));
```

`oopFactory::new_constantPool()` 的第一个入参是 `length`, 其在 `ClassFileParser::parse_constant_pool()` 函数中实现了对常量池大小的解析。`length` 值代表当前字节码文件的常量池中一共包含多少个常量池元素, 该数值由 Java 编译器在编译期间通过分析计算得出, 最终将其保存

在字节码文件中，所以 JVM 在解析常量池时可以直接拿来使用，这里的 `length` 便是 JVM 直接从字节码文件中读取出来的。在前面章节对 Java 字节码文件结构的分析中，讲到了常量池的大小的计算，有不清楚的小伙伴可以先温习前面的章节。不过要注意的是，`length` 并不表示常量池需要占用多大的内存空间，而是代表一共有多少个常量池元素。有了这个值，JVM 就能对常量池中的常量池元素进行逐个遍历处理。

`oopFactory::new_constantPool` 链路比较长，下面先给出总体调用路径（如图 5.3 所示）。

由图 5.3 可以看出，常量池内存分配的链路比较长，下面我们先对这条链路所涉及的类型进行一个简单说明：

- ◎ **oopFactory**。顾名思义，就是 oop 的工厂类。工厂类设计模式主要负责生产专门的对象，与具体的编程语言无关。前文讲过，Java 语言是一门面向对象的语言，所有一切皆对象，这种面向对象的特性不仅在语法层面贯彻得很彻底，而且在 JVM 内部实现层面也得到彻底的实现。在 JVM 内部，常量池、字段、符号、方法等一切都被对象包装起来，所有这一切对象在内存中都通过 oop 这种指针进行跟踪（指向），JVM 根据 oop 所指向的实际内存位置便可获取到对象的具体信息。而在 JVM 内部，所有这些对象的内存分配、对象创建与初始化（清零）工作都通过 `oopFactory` 这个入口得以统一实现。本节所讲解的常量池也不例外。
- ◎ **constantPoolKlass**。常量池类型对象。对于每一种对象，JVM 内部都通过 oop 指针指向到某个内存位置，这个内存位置往往便是与该 oop 相对应的 `klass` 类型。`klass` 用于描述 JVM 内部一个具体对象的结构信息，例如，一个 Java 类中包含哪些字段、哪些方法、哪些常量。同理，常量池在 JVM 内部也被表示为一种对象，虽然开发者并不能在源程序中通过 Java 程序来表示它。不同的 Java 类被编译后所生成的字节码文件中的常量池大小、元素顺序和结构等都不相同，因此 JVM 内部必须要预留一段内存区块来描述常量池的结构信息，这便是 `constantPoolKlass` 的意义所在。后文还会对其进行专门讲解，尤其是最重要的与源码层面的 Java 类相关联的相关 `klass` 对象。
- ◎ **collectedHeap**。闻声辨人，顾名思义，这表示 JVM 内部的堆内存区，可被垃圾收集器回收并反复利用。其代表 JVM 内部广义的堆内存区域，在 JDK 6 时代，这个内存区域会包含用于分配 Java 类对象实例的堆内存区、常量池区和 `perm` 区（方法区）。在 JVM 内部，除了堆栈变量之外的一切内存分配，都需要经过本区域。如果 JVM 内部的一个实例对象不在这一区域申请内存空间，那么只能跑到 JVM 堆外内存去申请了。

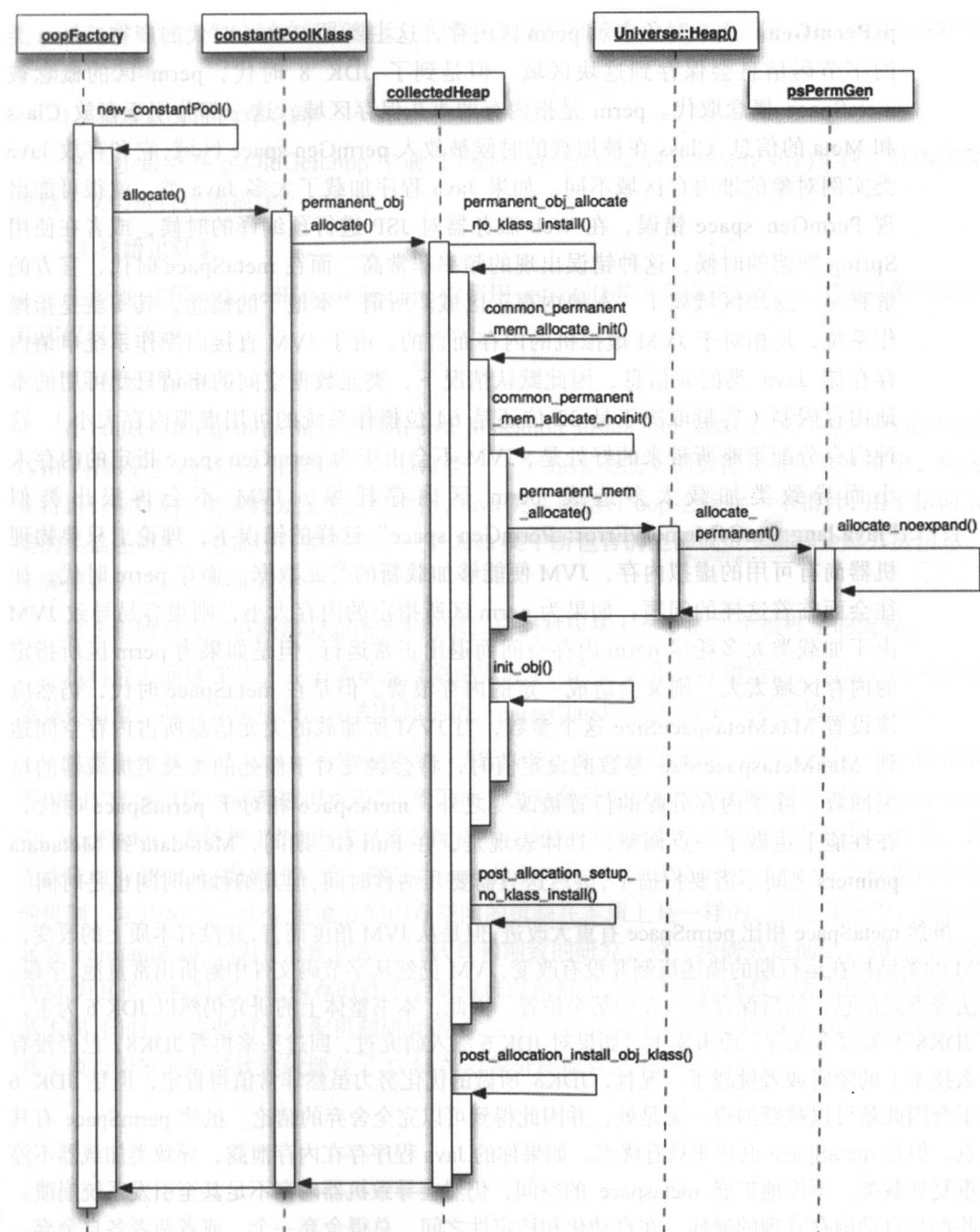


图 5.3 常量池内存分配调用链路

- ◎ **psPermGen**。这个对象表示 perm 区内存，这主要是 JDK 6 时代的产物，Java 类的字节码信息会保存在这块区域。但是到了 JDK 8 时代，perm 区的概念被 metaSpace 概念取代。perm 是指内存的永久保存区域，这一部分用于存放 Class 和 Meta 的信息，Class 在被加载的时候被放入 permGen space 区域，它和存放 Java 类实例对象的堆内存区域不同，如果 Java 程序加载了太多 Java 类，就很可能出现 PermGen space 错误，在 Web 服务器对 JSP 进行预编译的时候，或者在使用 Spring 框架的时候，这种错误出现的频率非常高。而在 metaSpace 时代，官方的解释是：这块区域属于“本地内存”区域。所谓“本地”的概念，其实就是指操作系统，是相对于 JVM 虚拟机的内存而言的。由于 JVM 直接向操作系统申请内存存储 Java 类的元信息，因此默认情况下，类元数据空间的申请只受可用的本地内存限制（容量取决于 32 位还是 64 位操作系统的可用虚拟内存大小）。这种内存分配策略所带来的好处是，JVM 不会由于为 permGen space 指定的内存太小而导致类加载太多造成 perm 区内存耗尽，JVM 不会再报出类似“java.lang.OutOfMemoryError: PermGen space”这样的错误了，理论上只要物理机器尚有可用的虚拟内存，JVM 便能够加载新的类元数据。而在 perm 时代，往往会面临着这样的问题：如果为 perm 区所指定的内存太小，则很容易导致 JVM 由于加载类太多耗尽 perm 内存空间而退出正常运行。但是如果为 perm 区所指定的内存区域太大，则又会造成一定的内存浪费。但是在 metaSpace 时代，仍然应该设置 MaxMetaspaceSize 这个参数，当 JVM 所加载的类元信息所占内存空间达到 MaxMetaspaceSize 参数的设定值时，将会触发对于僵死的类及类加载器的垃圾回收。除了内存分配的位置被改变之外，metaSpace 相对于 permSpace 时代，在性能上也做了一点调整，具体表现是，在 Full GC 期间，Metadata 到 Metadata pointers 之间不需要扫描了，虽然这只需要几纳秒时间，但几纳秒的时间也是时间。

虽然 metaSpace 相比 permSpace 有重大改进，但是从 JVM 角度而言，并没有本质上的改变，JVM 的类结构在运行期的描述机制并没有改变，JVM 仍然从字节码文件中解析出常量池、字段、方法等类元信息，然后保存到内存中某个位置。因此，本书整体上的研究仍然以 JDK 6 为主，对 JDK8 不做过多深究。而事实上，如果对 JDK 6 深入研究过，回过头来再看 JDK8，已经没有什么技术上的障碍或者挑战了。况且，JDK8 所做的优化努力虽然非常值得肯定，但是 JDK 6 并不会因此就可以被贬抑得一无是处，并因此得到可以完全舍弃的结论。虽然 permSpace 有其缺点，但是 metaSpace 也并非只有优点。如果你的 Java 程序存在内存泄露，导致类加载器不停地重复加载类，不停地扩展 metaspace 的空间，仍然会导致机器内存不足甚至引发系统崩溃。这些都是自动内存管理的通病，在自动化和稳定性之间，总得舍弃一个，或者两者各自舍弃一点以达到某种均衡。

虽然常量池内存分配的链路很长,但是从宏观层面来看,常量池内存分配大体上可以分为下面3个步骤。

1) 在堆区分配内存空间

这一步最终在 `psOldGen.hpp` 中通过 `object_space()->allocate(word_size)` 实现。具体实现的思路下文会进行详细描述。

2) 初始化对象

主要通过 `collectedHeap.inline.hpp` 中调用 `init_obj()` 进行对象初始化。所谓的对象初始化,其实仅仅是清零。

3) 初始化 oop

在 `collectedHeap.inline.hpp` 中调用 `post_allocation_install_obj_klass()` 完成 oop 初始化并赋值。JVM 内部通过 oop-klass 来描述一个 Java 类。一个 Java 类的实例数据会被存放在堆中,而为了支持运行期反射、虚函数分发等高级操作,Java 类实例指针 oop 会保存一个指针,用于指向 Java 类的类描述对象,类描述对象中保存一个 Java 类中所包含的全部成员变量和全部方法信息。本步骤便是为这一目标而设计的。

这3步所对应的链路部分如图5.4所示,分别对应从上至下的3个被框定的流程。

执行完上面这3步,一个常量池对象便完成了内存分配和部分初始化,但是此时常量池对象是空对象,其内存区还没有填充具体的值, `parseClassFile()` 函数下一步会做这个事情。

这里额外插一句,Java 字节码中的一切皆对象,无论是常量池、成员变量、方法还是数组等,在 JVM 虚拟机内部都被识别为“对象”,而为类对象分配内存空间的操作都封装在 `oopFactory` 中。`oopFactory` 为各种 JVM 内建对象分配内存空间并初始化类型实例的机制还是一样的,都是先获取对应的 `klass` 类描述对象,然后为 oop 分配内存空间。JVM 为一个 Java 类分配内存空间的机制,与 JVM 为一个常量池分配内存空间的机制在本质上是一样的,因此在这里,我们将一起对 `constantPool` 的内存分配机制进行认真和细致的研究,这一块研究透彻了,到了后面看到 JVM 为其他各种对象分配内存时就不会觉得难以理解了,主要的原理都是类似的,只有部分细节不同。同时,将这些对象分配机制研究透了之后,可以反过来促进对 JVM 的执行引擎的研究,其实执行引擎的很多伏笔就埋在了 JVM 类对象分配流程之中。

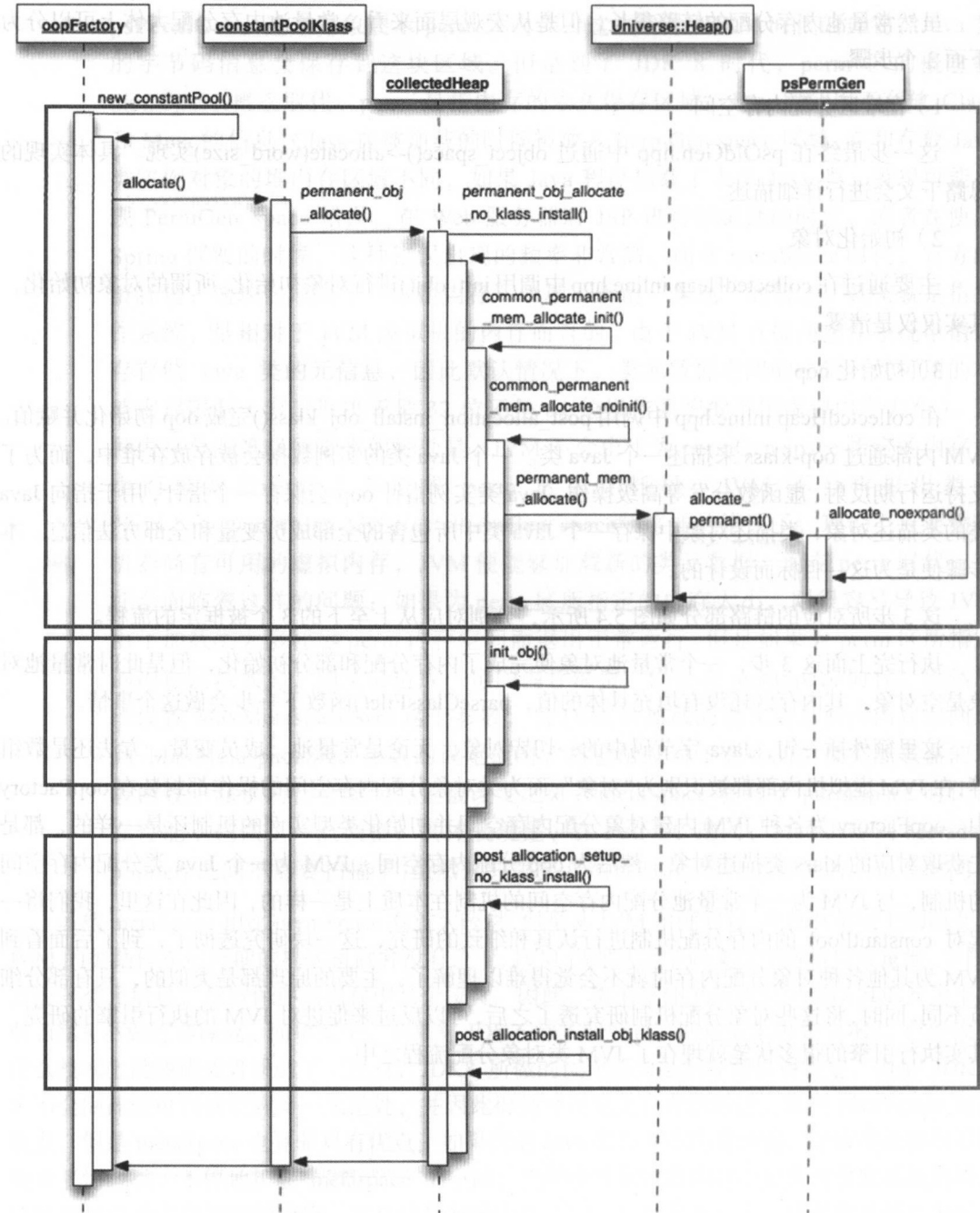


图 5.4 常量池内存分配三大步

下面分别对 constantPool 对象初始化的各主要步骤进行详细讲解。

5.1.2 内存分配

从 oopFactory::new_constantPool()调用开始一直到 mutableSpace::allocate(),这个过程可以认为是第一个阶段,即为 constantPool 申请内存。

前面讲过,内存申请主要关注两点:一是应该申请多大的内存;二是在哪里申请内存。在图 5.5 所示的链路中,从一开始便知道要申请多大的内存,内存的大小就是 length 变量所代表的值。因此,常量池有多少个常量池元素,最终便会分配多大的内存,至于为何是这样,下面会详细讲解。下面先分析 JVM 为 constantPool 申请内存的机制。

这条链路所涉及的接口调用及其所在的文件位置如图 5.5 所示。

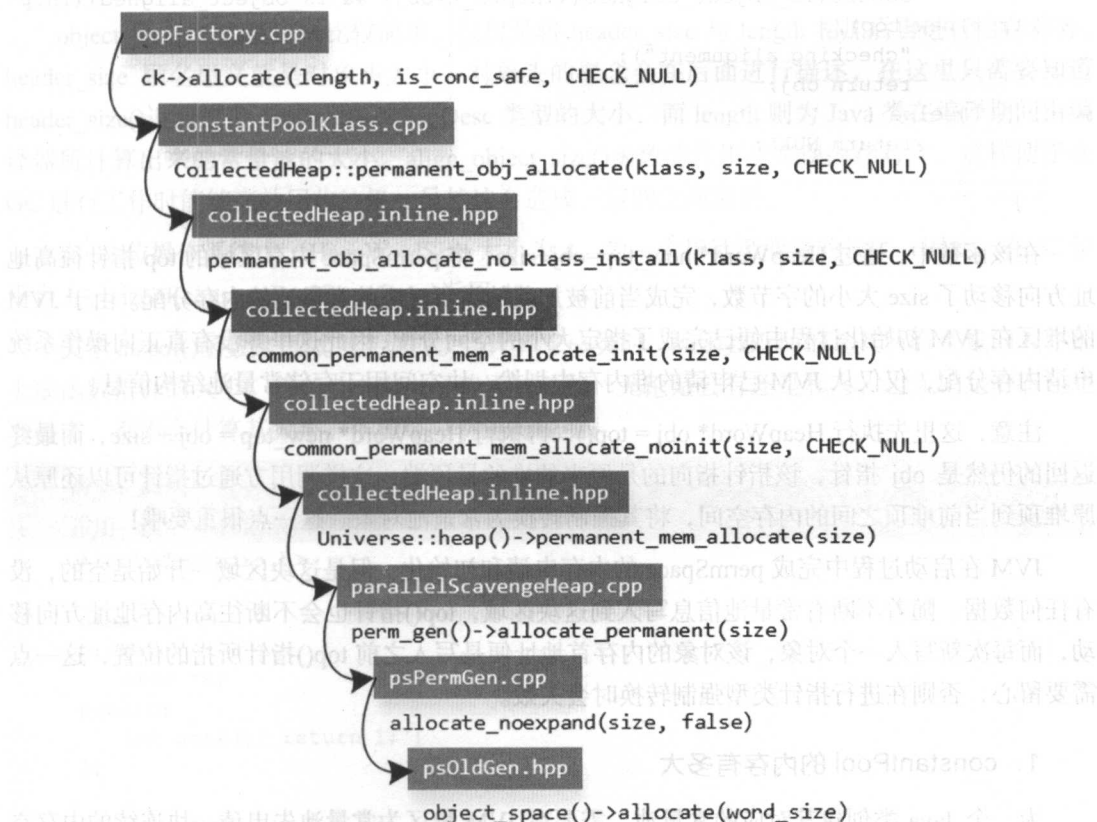


图 5.5 JVM 为常量池对象 constantPool 分配内存的链路

内存申请最终通过 `object_space()->allocate()` 实现，其上游一系列链路主要实现方法的逐级调用，为最后调用 `allocate()` 作铺垫。`allocate()` 函数定义在 `mutableSpace.cpp` 中，定义如下：

清单：src/share/vm/gc_implementation/shared/mutableSpace.cpp

作用：在 JVM 堆区分配内存

```
// This version requires locking. */
HeapWord* MutableSpace::allocate(size_t size) {
    assert(Heap_lock->owned_by_self() ||
        (SafepointSynchronize::is_at_safepoint() &&
         Thread::current()->is_VM_thread()),
        "not locked");
    HeapWord* obj = top();
    if (pointer_delta(end(), obj) >= size) {
        HeapWord* new_top = obj + size;
        set_top(new_top);
        assert(is_object_aligned((intptr_t)obj) && is_object_aligned((intptr_t)new_top),
            "checking alignment");
        return obj;
    } else {
        return NULL;
    }
}
```

在该函数中，通过 `HeapWord* new_top=obj+size`，将 `permSpace` 内存区域的 `top` 指针往高地址方向移动了 `size` 大小的字节数，完成当前被加载的类所对应的常量池的内存分配。由于 JVM 的堆区在 JVM 初始化过程中便已完成了指定大小的空间分配，因此这里并没有真正向操作系统申请内存分配，仅仅从 JVM 已申请的堆内存中划拨一块空间用于存储常量池结构信息。

注意，这里先执行 `HeapWord* obj = top()`，再执行 `HeapWord* new_top = obj + size`，而最终返回的仍然是 `obj` 指针，该指针指向的是原来的堆的最顶端，这样调用方通过指针可以还原从原堆顶到当前堆顶之间的内存空间，将其强制转换为常量池对象。这一点很重要哦！

JVM 在启动过程中完成 `permSpace` 的内存申请和初始化，但是这块区域一开始是空的，没有任何数据。随着不断有常量池信息写入到这块区域，`top()` 指针也会不断往高内存地址方向移动，而每次新写入一个对象，该对象的内存首地址便是写入之前 `top()` 指针所指的位置，这一点需要留心，否则在进行指针类型强制转换时会失败。

1. constantPool 的内存有多大

为一个 Java 类创建其对应的常量池，需要在 JVM 堆区为常量池先申请一块连续的内存空间。所申请的内存空间的大小取决于一个 Java 类在编译时所确定的常量池大小，更直白地说，

就是取决于你所定义的类的大小。在上面所描绘的常量池初始化调用链路中，可以看到 `size` 这个变量一直从链路前端被透传到最末端，这个 `size` 变量便是 JVM 为当前常量池所计算出来的内存大小。

在常量池初始化链路中会调用 `constantPoolKlass::allocate()` 方法，该方法会调用 `constantPoolOopDesc::object_size(length)` 方法来获取常量池大小，该方法的原型如下：

清单：src/share/vm/oops/constantPoolOop.hpp

作用：计算 Java 类所对应的常量池的内存大小

```
static int object_size(int length) {
    return align_object_size(header_size() + length);
}

static int header_size() {
    return sizeof(constantPoolOopDesc) / HeapWordSize;
}
```

`object_size()`的实现逻辑比较简单，仅仅是将 `header_size` 与 `length` 相加后再进行内存对齐。`header_size` 顾名思义就是对象头大小，对象头的概念会在后面进行描述，在这里只需要知道 `header_size()`返回的是 `constantPoolOopDesc` 类型的大小，而 `length` 则为 Java 类在编译期间由编译器所计算出来的常量池的大小。`align_object_size()`函数的作用是实现内存对齐，这样便于在 GC 进行工作时能够高效回收垃圾，虽然这会造成一定的空间浪费。

在 32 位操作系统上，`HeapWordSize` 大小为 4，为一个指针型变量的长度。因此，在 32 位平台上，`sizeof(constantPoolOopDesc)`返回 40。

关于 `sizeof()`函数这里做一点说明。当计算 C++类时，该函数返回的是其所有变量的大小加上虚函数指针的大小。若在类中定义了普通的函数，无论是公有还是私有，也无论是静态还是非静态，都不会计算其大小。下面举一例进行说明：

清单：测试

作用：演示 `sizeof()`函数的功能

```
#include<stdio.h>
```

```
class A{
private:
    char *a;
public:
    int getA(){ return 1; }
};
```

```
int main(){
    printf("sizeof(A)=%d\n", sizeof(A));
```

```

    return 0;
}

```

在 32 位平台上运行该程序，最终打印出来的值是 4，因为 A 类中只包含一个指针型变量，而 32 位平台上一个指针的数据宽度是 4，所以最终打印出来 4。

根据这个演示程序，可以推算 `sizeof(constantPoolOopDesc)` 返回值的大小。`constantPoolOopDesc` 本身包含 8 个字段，如下：

```

typeArrayOop    _tags;
constantPoolCacheOop _cache;
klassOop        _pool_holder;
typeArrayOop    _operands;
int             _flags;
int             _length;
volatile bool    _is_conc_safe;
int             _orig_length;

```

由于 `constantPoolOopDesc` 继承自 `oopDesc` 父类，因此 `constantPoolOopDesc` 类还会包含来自父类的 2 个成员变量：

```

volatile markOop _mark;
union _metadata {
    wideKlassOop    _klass;
    narrowOop        _compressed_klass;
} _metadata;

```

注意：`markOop` 的类型原型是 `markOopDesc*` 指针类型，因此 `_mark` 的类型是指针，在 32 位平台上占 4 字节的内存。`_metadata` 是联合体，联合体内部是指针类型，因此也占 4 字节空间。

虽然 `oopDesc` 类内部包含 `static BarrierSet* bs` 这样一个变量，但是这是静态类型的变量，在 JVM 启动之初其会被操作系统直接分配到 JVM 程序的数据段内存区，在 JVM 为 Java 程序分配内存时，不会为 `_bs` 这样的全局变量在 JVM 堆内存或者 `permSpace` 内存区另外分配空间。

如此看来，`constantPoolOopDesc` 最终实际包含 10 个字段，因此在 32 位平台上，`sizeof(constantPoolOopDesc)` 将返回 40。各位小伙伴大可以使用 GDB 在 32 位平台上断点调试，在断点时直接打印表达式 `sizeof(constantPoolOopDesc)` 的值进行验证。

搞清楚了 `sizeof(constantPoolOopDesc)`，还需要研究下 `HeapWordSize`，因为通过 `header_size()` 计算 `constantPoolOopDesc` 类大小时使用到了 `HeapWordSize`（`header_size()` 的计算公式是 `sizeof(constantPoolOopDesc)/HeapWordSize`）。

`HeapWordSize` 定义如下：

清单: src/share/vm/utilities/globalDefinitions.hpp

作用: 计算 HeapWordSize 大小

```
const int HeapWordSize      = sizeof(HeapWord);

class HeapWord {
    friend class VMStructs;
private:
    char* i;
#ifdef PRODUCT
public:
    char* value() { return i; }
#endif
};
```

可以看到, HeapWordSize 是 HeapWord 类的大小, 而该类只包含一个 char* 指针型变量, 因此在 32 位平台上, sizeof(HeapWordSize) 返回 4, 即其大小是 4 字节大小。如果在 64 位平台上, sizeof(HeapWordSize) 返回 8, 即其大小是 8 字节大小。所以, HeapWord 的大小其实就是一个指针的大小, 不同平台的指针所占的内存大小是不同的。同理, header_size() 函数返回的也是当前平台上的指针大小。

至此, header_size() 函数的作用已经十分清楚了, sizeof(constantPoolOopDesc) 返回 constantPoolOopDesc 类型本身所占内存的总字节数, 而 HeapWordSize 则返回对应平台上的一个指针宽度。在 32 位平台上, 一个指针宽度为 4 字节, 即双字(双字占 4 字节), 因此 header_size() 计算出的结果表示 constantPoolOopDesc 这个类型实例在内存中所占用的双字数。

理解了上面所讲的 header_size() 函数的含义之后, 再回过头来看 object_size(int length) { return align_object_size(header_size() + length); } 这个函数, 便很容易理解该函数的思路了, 其实 object_size() 函数最终返回的值代表 constantPoolOopDesc 类型本身的大小加上常量池的大小 length, constantPoolKlass::allocate() 便根据这个结果, 从 JVM 的 perm 区申请所需的内存空间大小。

最终, constantPoolKlass::allocate() 从 JVM 堆内存中所申请的内存空间大小包含如下所示的两部分:

constantPoolOopDesc 大小

Java 类常量池元素数量

代码总是枯燥的, 因此我们还是以前文所举的 Iphone6s.Java 为例。该类定义如下:

清单: Iphone6s.java

作用: Iphone6s 类型结构

```
public class Iphone6s {
    int length = 138;        // 长度 (毫米)
    int width = 67;         // 宽度 (毫米)
```

```
int height = 7;           //高度 (毫米)
int weight = 143;         //重量 (克)
int ram = 2;              //ram 容量 (G)
int rom = 16;             //rom 容量 (G)
int pixel = 1200;         //摄像头像素 (万)
}
```

使用十六进制编辑器打开编译后生成的字节码文件，得到常量池的长度 `length`，如图 5.6 所示。

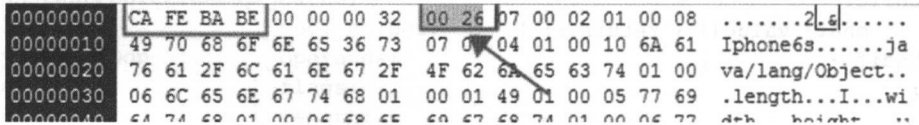


图 5.6 示例 Java 程序字节码文件中的常量池大小

图 5.6 中第一个方框内容是魔数，值为 `0xCAFEBA BE`，因此图中箭头所指的字节便代表常量池的元素数量。图中所示的常量池的元素数量为 `0x26`，换算成十进制为 38。由此可知，`Iphone6s.class` 字节码文件中的常量池一共包含 38 个元素。使用 `javap -verbose` 命令进行分析的结果如下：

```
>javap -verbose Iphone6s
Compiled from "Iphone6s.java"
public class Iphone6s extends java.lang.Object
  SourceFile: "Iphone6s.java"
  minor version: 0
  major version: 50
  Constant pool:
    const #1 = class      #2;      // Iphone6s
    const #2 = Asciz      Iphone6s;
    const #3 = class      #4;      // java/lang/Object
    const #4 = Asciz      java/lang/Object;
    const #5 = Asciz      length;
    const #6 = Asciz      I;
    //.....
    const #34 = Asciz      this;
    const #35 = Asciz      LIphone6s;;
    const #36 = Asciz      SourceFile;
    const #37 = Asciz      Iphone6s.java;
```

`javap` 命令所分析出的结果一共有 37 个常量池元素，之所以比字节码文件中的少一个，是因为 JVM 会保留第 0 号常量池位置，因此仅从第 1 个开始计算。如此便能保持一致。

如果 JVM 当前正在解析 `Iphone6s.class` 字节码文件的常量池信息，那么按照上面的分析，最终 JVM 会从 `permSpace` 中划分出 $(40 + 38) * 4$ 字节的内存大小（32 位平台）。

现在问题就来了,为什么 JVM 为常量池对象分配这么大的内存呢?

要解答这个疑问,需要知道 JVM 是如何解析字节码的常量池信息的,后文会专门讲解细节,因此这里先将问题放下,先讨论一个重要的问题:常量池的内存布局。

2. 内存空间布局

在为 `constantPoolOop` 常量池对象分配内存时,需要分析 JVM 为常量池所申请的内存空间布局的模型。刚才提到, JVM 为 `constantPoolOop` 实例对象所分配的内存空间大小是 $(\text{headSize} + \text{length})$ 个指针宽度(或者双字宽度)。JVM 为常量池对象申请的内存位于 perm 区, perm 区本身是一片连续的内存区域,而 JVM 为常量池申请内存时也是整片区域连续划分,因此每一个 `constantPoolOop` 对象实例在 perm 区中都是连续分布的,不会存在碎片化。

最终申请好的内存空间布局如图 5.7 所示。

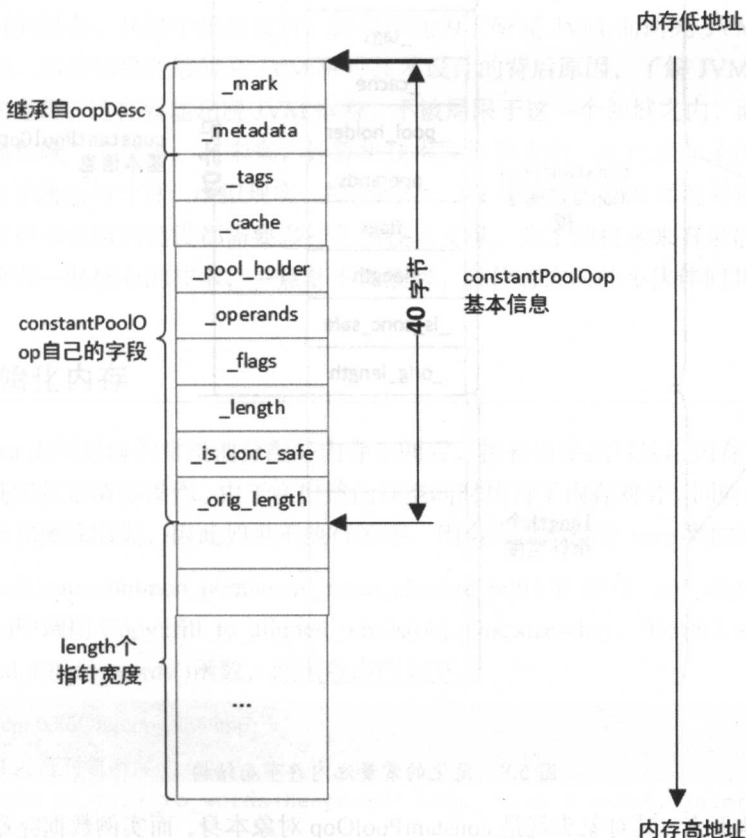


图 5.7 `constantPoolOop` 的内存布局

注：该图假设 JVM 运行于 Linux 32 位平台上，因此指针宽度为 4 字节，并且常量池分配方向为从低地址内存往高地址内存方向。

图 5.7 看起来虽然简单，但是在 JVM 内部，几乎所有的对象都是这种布局。总体而言，JVM 内部为对象分配内存时，会先分配对象头，然后分配对象的实例数据，不管字段对象还是方法对象，抑或是数组，莫不如此。再强调一遍，JVM 内部为对象实例分配内存空间的模型都是“对象头+实例数据”的结构哟！先用这个模型对图 5.7 所示的 `constantPoolOop` 的内存布局进行简化，得到如图 5.8 所示的布局。

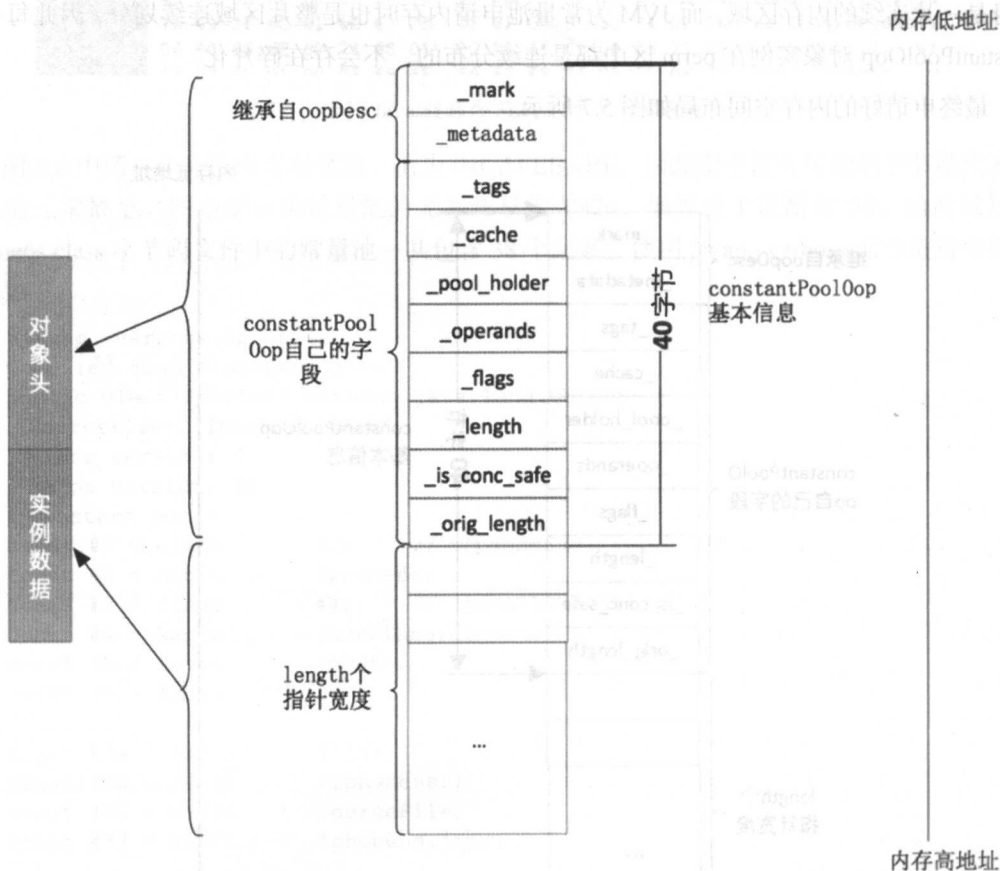


图 5.8 简化的常量池内存布局结构

对于常量池而言，其对象头就是 `constantPoolOop` 对象本身，而实例数据究竟放啥暂时还不知道（原谅我无法预先进行剧透），不过聪明的你已经知道了实例数据所占空间的大小既然等于

Java 字节码文件中所有常量池元素所占空间的大小,那么可以大胆猜测,实例数据应该就是保存 Java 字节码文件中的常量池元素的某些特殊信息,但是常量池信息量往往比其数量要大得多,因此紧跟在 constantPoolOop 之后的这一块实例数据区不可能用于存储常量池的全部信息。事实上,这一块的存储机制是比较复杂的,后续的 2 个小节将会专门讲述这个问题。目前只需要知道常量池的内存布局总体上也是由对象头和实例数据这两块组成的。

有句话是怎么说的?知其然,更要知其所以然。虽然如此浓墨重彩地深入详细地分析区区一个常量池的内存模型是很必要的,但是本书并不打算仅仅讲解 what 和 how,后面会分析为何 JVM 要将常量池分配在 perm 区 (JDK 1.6),同时需要考虑一个最核心的问题:JVM 为何要设计常量池这样一种机制。这两个问题是 why,研究 what 和 how 通常仍然属于技术的范畴,而研究 why 则意味着超脱技术而上升到哲学的高度,虽然这里所谓的哲学也许被称为“技术哲学”要显得更加合理。作为技术爱好者,知其然固然好,能够做到知其所以然更是难能可贵,但是一旦做到了,所得到的回报往往会超出想象。以 JVM 为例,虽然绝大多数开发者也许一辈子都没有开发虚拟机的机会,从这个角度而言,如果单纯为了研究 JVM 而研究 JVM,则一定是在浪费生命和时间。但是如果能够研究 JVM 种种技术设计的背后原因,了解 JVM 的各个模块之所以这样设计的道理,这样就能超脱 JVM 本身,不被局限于这一个领域之内,而能达到“天高任鸟飞,海阔凭鱼跃”之境界。在未来,计算机技术发展的方向一定比现如今还要更为宽广,机器人领域、量子通信与计算、虚拟现实、神经网络、3D 打印等各方面都需要较为扎实的基本功,一座座未来技术大厦的建设都需要多样化的技术支撑,为了迎接未来有可能出现的技术潮流,现在深入研究一些核心的技术,一点都不算浪费。各位有志向的小伙伴们共勉!

5.1.3 初始化内存

JVM 为 Java 类所对应的常量池分配好内存空间后,接着需要执行这段内存空间的初始化。所谓的初始化其实就是清零操作。由于在申请内存空间时执行了内存对齐,同时由于 JVM 堆区会反复加载新类和擦除旧类,因此如果不执行清零,则会影响后续对 Java 类的解析。

在 CollectedHeap::common_permanent_mem_allocate_init() 中调用 init_obj(obj, size), 在 init_obj(obj, size) 中调用 Copy::fill_to_aligned_words(obj + hs, size - hs), 在 x86 Linux 平台上, 该函数最终调用 pd_fill_to_words() 函数, 此函数声明如下:

清单: /src/cpu/x86/vm/copy.x86.hpp

作用: JVM 内部对象清零

```
static void pd_fill_to_words(HeapWord* tohw, size_t count, jint value) {
#ifdef AMD64
    jlong* to = (jlong*) tohw;
```

```

    jlong v = ((julong) value << 32) | value;
    while (count-- > 0) {
        *to++ = v;
    }
#else
    jint* to = (jint*)tohw;
    count *= HeapWordSize / BytesPerInt;
    while (count-- > 0) {
        *to++ = value;
    }
#endif // AMD64
}

```

在 `pd_fill_to_words()` 函数中，会将指定内存区的内存数据全部清空为零值，也即该函数的第 3 个入参 `value` 值。由于在 `CollectedHeap::init_obj()` 中调用了 `Copy::fill_to_aligned_words(obj + hs, size - hs` 函数，)而 `Copy::fill_to_aligned_words()` 函数实际有 3 个入参，声明如下：

清单：/src/share/vm/utilities/copy.hpp

作用：清零

```

static void fill_to_aligned_words(HeapWord* to, size_t count, jint value =
0) {
    assert_params_aligned(to);
    pd_fill_to_aligned_words(to, count, value);
}

```

注意，该函数第 3 个参数默认为 0，因此最终在执行 `pd_fill_to_words()` 函数时，指定的内存区会全部被抹为零值。

5.2 oop-klass 模型

现在，JVM 已经为 `constantPoolOop` 分配好内存并进行清零，接下来会进行非常重要的一步：填充内存。JVM 为 `constantPoolOop` 申请内存，随后会解析 Java 字节码中的常量池信息，并将解析的中间结果暂存到所申请的内存中去。但是在讲解 `constantPoolOop` 的解析过程之前，必须要先弄清楚一个概念——`oop-klass` 一分为二的内存模型。上文讲过，JVM 为常量池所分配的内存的布局包含两部分，分别是对象头和实例数据。当年詹爷使用了一种特殊的模型来表示类型，这种模型就是 `oop-klass`。因此，不把这种特殊的模型研究清楚，很难理解 JVM 为对象头分配内存的机制，接下来的源码阅读也一定会很吃力。

事实上，即使弄懂了 JVM 的类模型表示机制，这部分源代码阅读起来仍然十分吃力——。吃力的一个重要原因是指针以及基于指针的各种数据类型之间的强制转换，可能当年詹爷也是

深受指针之苦，因此才下定决心要创造出一个没有指针的编程世界。为了达到消灭“指针”的目的，JVM 本身的类模型变得格外复杂，这给源码解读带来相当大的困难。并且这种复杂性和基本的类模型从 JVM 发布以来一直没有经过大的变更，即使从 JDK6 到 JDK8，也仅仅是对 JVM 内部的几种具体的类型进行了重组和去繁就简，并没有进行根本上的变革。因此如果对 JDK6 的类模型研究通透，则对 JDK8 的类模型自然会触类旁通，一看就懂。不过最为关键的是，只要 Java 语言本身对外所提供的功能特性不发生巨大变化，则 JVM 内部的类模型也不会发生巨大的质变，这与本书一开始的章节描述 JVM 执行引擎的技术选择一样，都有其技术上的必然性，虽然真正实现上的细节可能会千差万别并会得到不断改进和优化，但是主要的算法与策略不会变化，一切都是由技术本身所决定的。

Java 是面向对象的编程语言，这种面向对象不仅体现在语法层面，也不仅仅体现在外在的 Java 类层面。JVM 在内部使用 C++ 类去描述 Java 类的面向对象特性，JVM 的奇特之处就在于，连同这些内部描述的类也被设计成面向对象机制，毫不夸张地说，JVM 内部的面向对象机制比外在的 Java 类语法层面的面向对象机制实现得更加彻底和更加纯粹，Java 语言所表现出来的面向对象特性与 JVM 内部的面向对象特性相比，简直有点小儿科。在 JVM 内部，不仅用于描述 Java 类的 C++ 对象被赋予相比于 C++ 语言本身所具备的面向对象特性更深一层的对象机制，而且其还用于描述相对于 Java 类外在对象而言属于 JVM 内部的不能由开发者控制的类型。JVM 内部用于描述 Java 字节码文件中的常量池类型不能被 Java 开发者访问和操作，但是即使是这种内部类，虽然本身使用 C++ 这种面向对象的语言描述，但是仍然被表示成 oop-klass 这种二分模型。关于 JVM 内部的这种一分为二的模型，完全可以专门开辟一个章节来详细阐述，但是如果仅仅阐述理论未免会流于空洞和枯燥，因此便将其安插在本章中，在描述 JVM 内部常量池的 oop-klass 表示机制的过程中，顺便对这种机制进行阐述。

5.2.1 两模型三维度

前文讲过，JVM 内部基于 oop-klass 模型描述一个 Java 类，将一个 Java 类一拆为二分别描述，第一个模型是 oop，第二个模型是 klass。所谓 oop，并不是 object-oriented programming（面向对象编程），而是 ordinary object pointer（普通对象指针），它用来表示对象的实例信息，看起来像个指针，而实际上对象实例数据都藏在指针所指向的内存首地址后面的一片内存区域中。而 klass 则包含元数据和方法信息，用来描述 Java 类或者 JVM 内部自带的 C++ 类型信息。其实，klass 便是前文一直在讲的数据结构，Java 类的继承信息、成员变量、静态变量、成员方法、构造函数等信息都在 klass 中保存，JVM 据此便可以在运行期反射出 Java 类的全部结构信息。当然，JVM 本身所定义的用于描述 Java 类的 C++ 类也使用 klass 去描述，这相当于使用另一种面向对象的机制去描述 C++ 类这种本身便是面向对象的数据。

JVM 使用 oop-klass 这种一分为二的模型描述一个 Java 类，虽然模型只有两种，但是其实从 3 个不同的维度对一个 Java 类进行了描述。侧重于描述 Java 类的实例数据的第一种模型 oop 主要为 Java 类生成一张“实例数据视图”，从数据维度描述一个 Java 类实例对象中各个属性在运行期的值。而第二种模型 klass 则又分别从两个维度去描述一个 Java 类，第一个维度是 Java 类的“元信息视图”，另一个维度则是虚函数列表，或者叫作方法分发规则。元信息视图为 JVM 在运行期呈现 Java 类的“全息”数据结构信息，这是 JVM 在运行期得以动态反射出类信息的基础。

下面的图 5.9 描述了 JVM 内部对 Java 类的“两模型三维度”的映射。

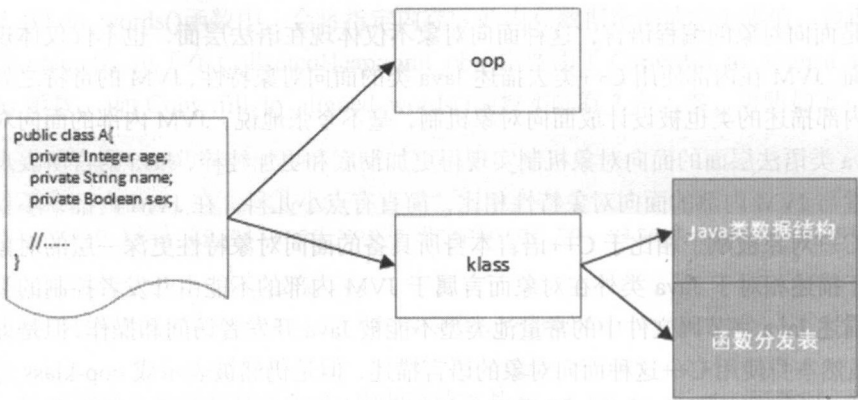


图 5.9 Java 类的两模型三维度

在 Java 编程语言中，并没有“虚函数”的概念，具体来说就是你不能在 Java 源代码中使用“virtual”这个关键字去修饰一个 Java 方法。而有过 C++ 编程经验的小伙伴都知道，C++ 实现面向对象多态性的关键字主要就是 virtual。这本来与 Java 毫无关系，毕竟是两种不同的语言，谁也无权强制要求别人为了支持多态就一定要通过“virtual”这个关键字。但是不巧的是，JVM 在内部使用 C++ 类所定义的一套对象机制去表达 Java 类的面向对象机制，这一表达顺手就把 Java 类的多态机制也包含在内了，毕竟 Java 号称是比 C++ 更加纯粹的面向对象的语言，结果总不能连个多态都不支持吧。但是詹爷非但不走寻常路，而且还把正常的路径给堵住了不让走，就是不让 Java 支持 virtual 的概念。但是这样就会带来一个问题，Java 类最终被表达成了 JVM 内部的 C++ 类，并且 Java 类方法的调用最终要通过对应的 C++ 类，但是 Java 语言是面向对象的，多态性是其基本特性，这意味着 JVM 内部的 C++ 类要能够支持 Java 语言的多态性，可是 Java 方法并不支持使用 virtual 这个关键字来修饰，这样问题就来了，C++ 层面怎样才能知道 Java 类中的哪个方法是虚函数，哪个方法不是虚函数呢？换言之，当面对一个多重继承的 Java 类体系时，JVM 内部的 C++ 类怎么才能将这种多态性表达出来呢？詹爷的做法很简单粗暴，那就是

将 Java 类的所有函数都视为是“virtual”的，这样 Java 类中的每个方法都可以直接被其子类、子子类覆盖而不需要增加任何关键字作为修饰符。正因为如此，Java 类中的每个方法都可以晚绑定，只不过对于一些确定的调用，在编译期便能实现早绑定。

正是因为 JVM 将 Java 类中的每一个函数都视为虚函数，所以最终在 JVM 内部的 C++ 层面，就必须维护一套函数分发表。关于函数分发表，没有 C++ 编程经验的小伙伴肯定不知其所云，不过不用着急，我们这里慢慢道来。

5.2.2 体系总览

在 JVM 内部定义了 3 种结构去描述一种类型：oop、class 和 handle 类。注意，这 3 种数据结构不仅能够描述外在的 Java 类，也能够描述 JVM 内在的 C++ 类型对象。

前面讲过，class 主要描述 Java 类和 JVM 内部 C++ 类型的元信息和虚函数，这些元信息的实际值就保存在 oop 里面。oop 中保存一个指针指向 class，这样在运行期 JVM 便能够知道每一个实例的数据结构和实际类型。handle 是对 oop 的行为的封装，在访问 Java 类时一定是通过 handle 内部指针得到 oop 实例的，再通过 oop 就能拿到 class，如此 handle 最终便能操纵 oop 的行为了（注意，如果是调用 JVM 内部 C++ 类型所对应的 oop 的函数，则不需要通过 handle 来中转，直接通过 oop 拿到指定的 class 便能实现）。class 不仅包含自己所固有的行为接口，而且也能够操作 Java 类的函数。由于 Java 函数在 JVM 内部都被表示成虚函数，因此 handle 模型其实就是 Java 类行为的表达。

先上一张图说明这种三角关系（如图 5.10 所示）。

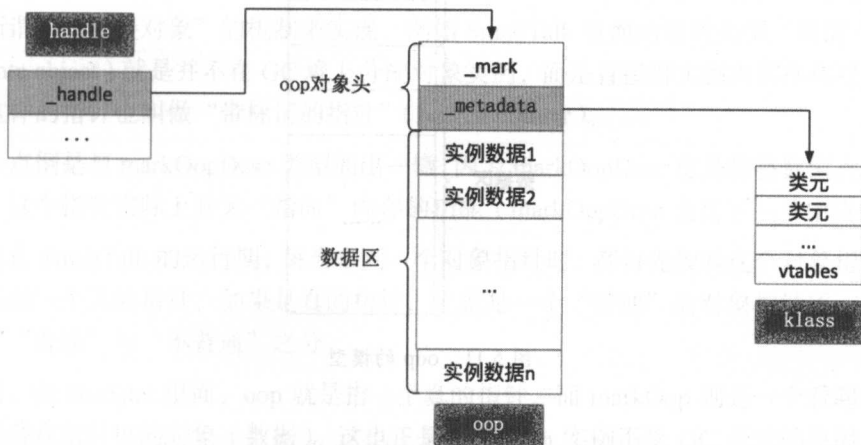


图 5.10 oop、class、handle 的三角关系

这种三角关系最终在数据结构中得以落地，在 `handles.hpp` 文件中定义了 `Handle` 类的基本结构：

清单：/src/share/vm/runtime/handles.hpp

作用：`Handle` 类的数据结构

```
class Handle VALUE_OBJ_CLASS_SPEC {  
private:  
    oop* _handle;  
  
    //.....  
};
```

可以看到，`Handle` 类内部只有一个成员变量 `handle`，该变量类型是 `oop*`，因此该变量最终指向的就是一个 `oop` 的首地址。换言之，只要能够拿到 `Handle` 对象，便能据此得到其所指向的 `oop` 对象实例，而通过 `oop` 对象实例又能进一步获取其所关联的 `klass` 实例，而获取到 `klass` 对象实例后，便能实现对 `oop` 对象方法的调用。因此，虽然从表面上看，`handle` 体系貌似是对 `oop` 的一种封装，但是实际上其醉翁之意在于最终的 `klass` 体系。

`oop` 一般由对象头、对象专有属性和数据体这 3 部分构成。其一般结构如图 5.11 所示。

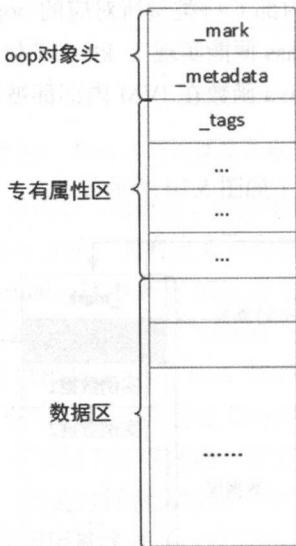


图 5.11 `oop` 的模型

例如,本节讲解的 JVM 内部对象 `constantPool` 常量池对象,其 oop 对象是 `constantPoolOop`,该对象的结构与上面的模型完全符合。上面这张 oop 模型图的中间布局是 oop 类型的专有属性区, JVM 内部定义了若干 oop 类型,每一种 oop 类型都有自己特有的数据结构, oop 的专有属性区便是用于存放各个 oop 所特有的数据结构的地方。

5.2.3 oop 体系

虽然前面我们已经接触过 `constantPoolOop`,也讲了很多 oop 的东西,但是 oop 究竟是啥?为什么要有这种模型?我们依然疑惑。

所谓 oop,就是 `ordinary object pointer`,也即普通对象指针。但是究竟什么才是普通对象指针呢?要搞清楚何谓 oop,要问 2 个问题:

1) HotSpot 里的 oop 指啥

Hotspot 里的 oop 其实就是 GC 所托管的指针,每一个 oop 都是一种 `xxxOopDesc*` 类型的指针。所有 `oopDesc` 及其子类(除神奇的 `markOopDesc` 外)的实例都由 GC 所管理,这才是最重要的,是 oop 区分 Hotspot 里所使用的其他指针类型的地方。

2) 对象指针之前为何要冠以“普通”二字

对象指针从本质上而言就是一个指针,指向 `xxxOopDesc` 的指针也是普通得不能再普通的指针,可是为何在 Hotspot 领域还要加一个“普通”来修饰?要回答这个问题,需要追溯到 OOP(这里的 OOP 是指面向对象编程)的鼻祖——SmallTalk 语言。

SmallTalk 语言里的对象也由 GC 来管理,但是 SmallTalk 里面的一些简单的值类型对象都会使用所谓的“直接对象”的机制来实现,例如 SmallTalk 里面的整数类型。所谓“直接对象”(immediate object)就是并不在 GC 堆上分配对象实例,而是直接将实例内容存在对象指针里的对象。这样的指针也叫做“带标记的指针”(tagged pointer)。

这一点倒是与 `markOopDesc` 类型如出一辙,因为 `markOopDesc` 也是将整数值直接存储在指针里面,这个指针实际上并无“指向”内存的功能(`markOopDesc` 会在下一节中讲解)。

所以在 SmallTalk 的运行期,每当拿到一个对象指针时,都得先校验这个对象指针是一个直接对象还是一个真的指针?如果是真的指针,它就是一个“普通”的对象指针了。这样对象指针就有了“普通”与“不普通”之分。

所以,在 HotSpot 里面,oop 就是指一个真的指针,而 `markOop` 则是一个看起来像指针但实际上是藏在指针里的对象(数据)。这也正是 `markOop` 实例不受 GC 托管的原因,因为只要出了函数作用域,指针变量就会直接被从堆栈上释放掉了,不需要垃圾回收了。

JVM 内部并不是 oop 一个人在战斗，而是整个一套继承体系在运作。在 oopsHierarchy.hpp 中定义了这个体系家族的所有成员：

```
清单：/src/share/vm/oops/oopsHierarchy.hpp
作用：oop 的继承体系

typedef class      oopDesc*                oop;
typedef class      instanceOopDesc*        instanceOop;
typedef class      methodOopDesc*          methodOop;
typedef class      constMethodOopDesc*     constMethodOop;
typedef class      methodDataOopDesc*      methodDataOop;
typedef class      arrayOopDesc*           arrayOop;
typedef class      objArrayOopDesc*        objArrayOop;
typedef class      typeArrayOopDesc*       typeArrayOop;
typedef class      constantPoolOopDesc*    constantPoolOop;
typedef class      constantPoolCacheOopDesc* constantPoolCacheOop;
typedef class      klassOopDesc*           klassOop;
typedef class      markOopDesc*            markOop;
typedef class      compiledICHolderOopDesc* compiledICHolderOop;
```

这些不同的 oop 类型能够分别描述不同的对象，具体作用见如表 5.1 所示。

表 5.1 oop 类型描述的对象

oop	所有 oop 的顶级父类
instanceOop	表示 Java 类实例
methodOop	表示 Java 方法
constMethodOop	表示 Java 方法中的只读信息（其实就是字节码指令）
methodDataOop	表示性能统计的相关数据
arrayOop	数组对象
objArrayOop	表示引用类型数组对象
typeArrayOop	表示基本类型数组对象
constantPoolOop	表示 Java 字节码文件中的常量池
constantPoolCacheOop	与 constantPoolOop 相伴生，是后者的缓存对象
klassOop	指向 JVM 内部的 klass 实例的对象
markOop	oop 的标记对象

面对这十几种不同的 oop 大可不必惊慌，事实上只需要关注两种最常用的即可：constantPoolOop 和 instanceOop。其中，constantPoolOop 已经在上一节中详细描述过，而 instanceOop 将会后面的章节详细描述。作为 Java 程序的解释器和虚拟运行介质，JVM 将 Java 实例映射成 instanceOop，这注定 instanceOop 是一个无法跳过的坎，当然也注定其过程一定很

精彩。

虽然 oop 类型比较多,但是只要深入研究一番 constantPoolOop 和 instanceOop 这两个最重要的 oop,其他的 oop 自会触类旁通。

在这里有必要再次对 JDK 的版本问题做个说明。可能很多 JVM 发烧友都知道, JDK 8 中的 oop 与 JDK 6 相比变化很大,整个继承体系都发生了变化,但是笔者认为本质上并没有发生多大变化。JDK 8 中仍然有描述 constantPool 常量池的 oop,仍然有描述 Java 类实例对象的 oop, JDK 8 并没有抛弃 Java 字节码文件中的常量池,没有对 Java 字节码文件结构进行大刀阔斧的调整(事实上这已经成为 Java 的一种标准,想改也改不了),并没有对描述 Java 类的 oop-class 这种二分模型进行根本上的改变。只要这些机制或标准不发生彻底的变化,那么相应的实现机制便注定不可能有本质上的不同。因此 JDK 8 虽然修改了继承关系,对一些 oop 进行了删减,但是在本质上与 JDK 6 仍然保持一致。正是由于这种原因,对 JDK 6 的研究并不会变得过时,更不会徒劳无功。

所以,本书基本以 JDK 6 源码为主,分析 JVM 的内存模型。如果小伙伴对 JDK 8 情有独钟,也完全不用担心辛辛苦苦做的研究会白费掉。

还有一点需要注意,由于 oopDesc 不同的子类类型名称都以 OopDesc 结尾,因此为了简化, JVM 为其取了别名,统一以 oop 结尾。

5.2.4 klass 体系

oop 的讲述先告一段落,再来看看 klass 部分。按照 JVM 的官方解释, klass 主要提供下面 2 种能力:

- ◎ klass 提供一个与 Java 类对等的 C++ 类型描述。
- ◎ klass 提供虚拟机内部的函数分发机制。

其实这种说法与上文所说的 2 种维度的含义是相同的。klass 分别从类结构和类行为这两方面去描述一个 Java 类(当然也包含 JVM 内部非开放的 C++ 类)。

与 oop 相同,在 JVM 内部也不是 klass 一个人在战斗,而是一个家族。klass 家族体系如下:

清单: /src/share/vm/klass/klassHierarchy.hpp

功能: klass 的继承体系

```
class Klass;
class instanceKlass;
class instanceMirrorKlass;
class instanceRefKlass;
```

```
class    methodKlass;
class    constMethodKlass;
class    methodDataKlass;
class    klassKlass;
class    instanceKlassKlass;
class    arrayKlassKlass;
class    objArrayKlassKlass;
class    typeArrayKlassKlass;
class    arrayKlass;
class    objArrayKlass;
class    typeArrayKlass;
class    constantPoolKlass;
class    constantPoolCacheKlass;
class    compiledICHolderKlass;
```

class 家族一看就比 oop 家族要人丁兴旺，让人望而生畏。但是不要紧，只要深入研究了其中两个最重要的 class，其他的便都会是浮云。

在 class 家族里，除去与 constantPoolOop 相对应的 constantPoolKlass 之外，最重要的莫过于 instanceKlass 和 klassKlass 了。这两个在后面的章节会陆续介绍，其中，instanceKlass，顾名思义，其实就是专门用于描述 Java 类的。名字取得好就这点好处，往往只需看个名字便能知其八分，绝对比看相的和算命的要准很多，所以本书中会使用很多次“顾名思义”，因为很多事情的真相其实就隐藏在变量名称或者类型的名称里面，无须太多解释。

class 家族的基类是 Klass 类，而 Klass 其实并不是顶级父类，Klass 继承了一个名叫 Klass_vtbl 的类，后者才是整个 class 家族的“元老”。当然，顾名思义，后者貌似是一个描述 vtbl（即虚函数表）的类，而事实上该类的确当此大任。

class 家族的各个成员的定位是不同的，具体如表 5.2 所示（由于 JDK 6 中的很多 class 在 JDK 8 中已经被删减了，因此这里仅挑重要的几个进行描述，有兴趣的小伙伴可以自行查阅 JDK 6 源码）。

表 5.2 class 家族成员

类	定 位
Klass	class 家族的基类
instanceKlass	虚拟机层面与 Java 类对等的数据结构
instanceMirrorKlass	描述 java.lang.Class 的实例
instanceRefKlass	描述 java.lang.ref.Reference 的子类
methodKlass	表示 Java 类的方法
constMethodKlass	描述 Java 类方法所对应的字节码指令信息的固有属性

续表

类	定 位
klassKlass	klass 链路的末端。 该类型在 JDK 8 中已经不复存在，但是由于其比较重要，因此本书依然会介绍这个特殊的 klass
arrayKlass	描述 Java 数组的信息，是个抽象基类
typeArrayKlass	描述 Java 中基本类型数组的数据结构
objArrayKlass	描述 Java 中引用类型数组的数据结构
constPoolKlass	描述 Java 字节码文件中的常量池的数据结构

由于 JDK 8 并没有对 JDK 6 进行本质上的模型变革，因此 JDK 6 中的这些重要的 klass 模型依然能够在 JDK 8 中找到，除了 klassKlass 这个特殊的类型。

既然 klass 能够描述 Java 类的数据结构（即元数据），那么一起来看看 klass 类里面究竟有些什么。基类 Klass 的定义如下：

```
清单：/src/share/vm/oops/klass.hpp
功能：Klass 的数据结构

class Klass : public Klass_vtbl {
protected:
    jint      _layout_helper;

    jint      _super_check_offset;

    Symbol*    _name;

public:
    oop* oop_block_beg() const { return adr_secondary_super_cache(); }
    oop* oop_block_end() const { return adr_next_sibling() + 1; }

protected:

    //=====oop block START=====//
    klassOop    _secondary_super_cache;

    objArrayOop _secondary_supers;

    klassOop    _primary_supers[_primary_super_limit];

    oop          _java_mirror;

    klassOop    _super;
```

```
classOop _subclass;

classOop _next_sibling;
//=====oop block END=====//

jint    _modifier_flags;
AccessFlags _access_flags;

juint    _alloc_count;

jlong    _last_biased_lock_bulk_revocation_time;
markOop  _prototype_header;
jint     _biased_lock_revocation_count;
}
```

该类包含的字段比较多，相关字段含义或作用如表 5.3 所示。

表 5.3 klass 类相关字段

字段名	含义/作用
_layout_helper	对象布局的综合描述符
_name	类名。例如，java.lang.String 类的 _name 属性值会是 java/lang/String
_java_mirror	类的镜像类
_super	父类
_subclass	指向第一个子类，若无，则为 NULL
_next_sibling	指向下一个兄弟节点，若无，则为 NULL
_modifier_flags	修饰符标识，例如 static
_access_flags	访问权限标识，例如 public

如果一个 Klass 既不是 instance 也不是 array，则其 _layout_helper 被设置为 0。如果 Klass 标识一个 instance，则其 _layout_helper 为正数，其值表示 instance 的大小。如果 Klass 代表的是一个数组，则 _layout_helper 为负数。

5.2.5 handle 体系

前面讲过，handle 封装了 oop，由于通过 oop 可以拿到 klass，而 klass 是对 Java 类数据结构和方法的描述，因此 handle 间接封装了 klass。JVM 内部使用一个 table 来存储 oop 指针。

如果说 oop 是对普通对象的直接引用，那么 handle 就是对普通对象的一种间接引用，中间

隔了一层。但是 JVM 内部为何要使用这种间接引用呢？答案是，这完全是为 GC 考虑。具体表现在 2 个地方：

- ◎ 通过 handle，能够让 GC 知道其内部代码都有哪些地方持有 GC 所管理的对象的引用，这只需要扫描 handle 所对应的 table，这样 JVM 便无须关注其内部到底哪些地方持有对普通对象的引用。
- ◎ 在 GC 过程中，如果发生了对象移动（例如从新生代移到了老一代），那么 JVM 的内部引用无须跟着更改为被移动对象的新地址，JVM 只需要更改 handle table 里对应的指针即可。

当然实际的 handle 作为对 Java 类方法的访问的包装，远不止上面所描述的这么简单。这里涉及 Java 类的类继承和接口继承的话题，在 C++ 领域，类的继承和多态性最终通过 vptr（虚函数表）来实现。在 klass 内部，记录了每一个类的 vptr 信息，具体而言分为两部分来描述。

1. vtable 虚函数表

vtable 中存放 Java 类中非静态和非 private 的方法入口，JVM 调用 Java 类的方法（非静态和非 private）时，最终会访问 vtable，找到对应的方法入口。

2. itable 接口函数表

itable 中存放 Java 类所实现的接口类方法。同样，JVM 调用接口方法时，最终会访问 itable，找到对应的接口方法入口。

不过要注意，vtable 和 itable 里面存放的并不是 Java 类方法和接口方法的直接入口，而是指向了 Method 对象入口，JVM 会通过 Method 最终拿到真正的 Java 类方法入口，得到方法所对应的字节码/二进制机器码并执行。当然，对于被 JIT 进行动态编译后的方法，JVM 最终拿到的是其对应的被编译后的本地方法的入口。

关于 vtable 和 itable 的话题先讲到这里，后面会单独开辟一个章节来详细讲解。

与 oop 和 klass 一样，在 JVM 内部，handle 也不是一个人在战斗，而是有一个庞大的家族。在 handles.hpp 文件中定义了 handle 体系的家族：

清单：/src/share/vm/runtime/handles.hpp

功能：handle 家族

//handle体系基类

```
class Handle VALUE_OBJ_CLASS_SPEC {
    //...
};
```

```
//KlassHandle基类
```

```
class KlassHandle: public Handle {
//...
};
```

```
//-----
```

```
//oop家族所对应的handle定义宏
```

```
#define DEF_HANDLE(type, is_a) \
class type##Handle; \
class type##Handle: public Handle { \
protected: \
type##Oop obj() const { return (type##Oop)Handle::obj(); } \
type##Oop non_null_obj() const { return (type##Oop)Handle::non_null_obj(); } \
public: \
/* Constructors */ \
type##Handle () : Handle() {} \
type##Handle (type##Oop obj) : Handle((oop)obj) { \
assert(SharedSkipVerify || is_null() || ((oop)obj)->is_a(), \
"illegal type"); \
} \
type##Handle (Thread* thread, type##Oop obj) : Handle(thread, (oop)obj) { \
assert(SharedSkipVerify || is_null() || ((oop)obj)->is_a(), "illegal type"); \
} \
/* Special constructor, use sparingly */ \
type##Handle (type##Oop *handle, bool dummy) : Handle((oop*)handle, dummy) {} \
/* Operators for ease of use */ \
type##Oop operator () () const { return obj(); } \
type##Oop operator -> () const { return non_null_obj(); } \
};
```

```
//定义oop家族的handle体系
```

```
DEF_HANDLE(instance, is_instance)
DEF_HANDLE(method, is_method)
DEF_HANDLE(constMethod, is_constMethod)
DEF_HANDLE(methodData, is_methodData)
DEF_HANDLE(array, is_array)
DEF_HANDLE(constantPool, is_constantPool)
DEF_HANDLE(constantPoolCache, is_constantPoolCache)
DEF_HANDLE(objArray, is_objArray)
DEF_HANDLE(typeArray, is_typeArray)
```

```

//-----
//class家族所对应的handle宏定义
#define DEF_CLASS_HANDLE(type, is_a) \
class type##Handle : public KlassHandle { \
public: \
    /* Constructors */ \
    type##Handle() : KlassHandle() {} \
    type##Handle (KlassOop obj) : KlassHandle(obj) { \
        assert(SharedSkipVerify || is_null() || obj->klass_part()->is_a(), \
            "illegal type"); \
    } \
    type##Handle (Thread* thread, KlassOop obj) : KlassHandle(thread, obj) { \
        assert(SharedSkipVerify || is_null() || obj->klass_part()->is_a(), \
            "illegal type"); \
    } \
\
    /* Access to klass part */ \
    type* operator -> () const { return (type*)obj()->klass_part(); } \
\
    static type##Handle cast(KlassHandle h) { return type##Handle(h()); } \
};

```

//定义klass家族的handle体系

```

DEF_CLASS_HANDLE(instanceKlass, oop_is_instance_slow )
DEF_CLASS_HANDLE(methodKlass, oop_is_method )
DEF_CLASS_HANDLE(constMethodKlass, oop_is_constMethod )
DEF_CLASS_HANDLE(klassKlass, oop_is_klass )
DEF_CLASS_HANDLE(arrayKlassKlass, oop_is_arrayKlass )
DEF_CLASS_HANDLE(objArrayKlassKlass, oop_is_objArrayKlass )
DEF_CLASS_HANDLE(typeArrayKlassKlass, oop_is_typeArrayKlass)
DEF_CLASS_HANDLE(arrayKlass, oop_is_array )
DEF_CLASS_HANDLE(typeArrayKlass, oop_is_typeArray_slow)
DEF_CLASS_HANDLE(objArrayKlass, oop_is_objArray_slow )
DEF_CLASS_HANDLE(constantPoolKlass, oop_is_constantPool )
DEF_CLASS_HANDLE(constantPoolCacheKlass, oop_is_constantPool )

```

可以看到，在 handles.hpp 中，通过 2 个宏分别批量声明了 oop 和 klass 家族的各个类所对应的 handle 类型。

在编译期，宏被替换后，便出现了如下 handle 体系（见表 5.4 和表 5.5）。

表 5.4 oop 体系所对应的 handle 体系

类	定 位
instanceHandle	类实例 handle

续表

类	定 位
methodHandle	方法实例 handle
constMethodHandle	方法字节码实例 handle
methodDataHandle	性能统计 handle
arrayHandle	数组 handle
constantPoolHandle	常量池 handle
constantPoolCacheHandle	常量池缓存 handle
objArrayHandle	引用类型数组 handle
typeArrayHandle	基本类型数组 handle

表 5.5 klass 体系所对应的 handle 体系

类	定 位
instanceKlassHandle	类元结构 handle
methodKlassHandle	方法元结构 handle
constMethodKlassHandle	方法固定类元结构 handle
klassKlassHandle	klass 链路末端类 handle
arrayKlassKlassHandle	描述基类数组元结构 handle
objArrayKlassKlassv	描述引用类型数组的类元结构 handle
typeArrayKlassKlasHandle	描述基本类型数组的类元结构 handle
arrayKlassHandle	基类数组元结构 handle
typeArrayKlassHandle	基本类型数组的类元结构 handle
objArrayKlassHandle	引用类型数组的类元结构 handle
constantPoolKlassHandle	常量池类元结构 handle
constantPoolKlassHandle	常量池缓存类元结构 handle

这里有个问题，前面不是一直在说 handle 是对 oop 的直接封装和对 klass 的间接封装吗，为什么这里却分别给 oop 和 klass 定义了 2 套不同的 handle 体系呢？这给人的感觉好像是，封装 oop 的 handle 和封装 klass 的 handle 并不是同一个 handle，既然不是同一个 handle，那么通过封装 oop 的 handle 还怎么去得到所对应的 klass 信息呢？

其实这正是 JVM 内部常常容易使人迷惑的地方。在 JVM 中，使用 oop-klass 这种一分为二的模型去描述 Java 类以及 JVM 内部的特殊类群体，为此 JVM 内部特定义了各种 oop 和 klass

类型。但是，对于每一个 oop，其实都是一个 C++ 类型，也即 klass；而对于每一个 klass 所对应的 class，在 JVM 内部又都会被封装成 oop。JVM 在具体描述一个类型时，会使用 oop 去存储这个类型的实例数据，并使用 klass 去存储这个类型的元数据和虚方法表。而当一个类型完成其生命周期后，JVM 会触发 GC 去回收，在回收时，既要回收一个类实例所对应的实例数据 oop，也要回收其所对应的元数据和虚方法表（当然，两者并不是同时回收，一个是堆区的垃圾回收，一个是永久区的垃圾回收）。为了让 GC 既能回收 oop 也能回收 klass，因此 oop 本身被封装成了 oop，而 klass 也被封装成 oop。而 JVM 内部恰好将描述类实例的 oop 全都定义成类名以 oop 结尾的类，并将描述类结构和方法信息的 klass 全都定义成类名以 klass 结尾的类，而 JVM 内部描述类信息的模型恰巧也叫作 oop-klass，与类名存在重合，这就导致了很多人的疑惑，这些疑惑完全是因为叫法上的重合而产生。

因此为了进一步解开疑惑，我们不妨换个叫法，不再将 JVM 内部描述类信息的模型叫作 oop-klass，而是叫作 data-meta 模型（瞎取的，名字没啥特殊含义）。然后将 JVM 内部的 oop 体系的类名全都改成以 Data 结尾，例如，methodData、instanceData、constantPoolData 等，同时将 klass 体系的类名也全都改成以 Meta 结尾，例如，methodMeta、instanceMeta、constantPoolMeta 等。JVM 在进行 GC 时，既要回收 Data 类实例，也要回收 Meta 类实例，为了让 GC 便于回收，因此对于每一个 Data 类和每一个 Meta 类，JVM 在内部都将其封装成了 oop 模型。对于 Data 类，其内存布局是前面为 oop 对象头，后面紧跟实例数据；而对 Meta 类，其内存布局是前面为 oop 对象头，后面紧跟实例数据和虚方法表。封装成 oop 之后，再进一步使用 handle 来封装，于是便有利于 GC 内存回收。

在这种新的模型中，不管是 Data 类还是 Meta 类，都是一种普通的 C++ 类型，只不过它们从不同的角度对 Java 类进行了描述。不管是 Data 类还是 Meta 类，当其所在的 JVM 的内存区域爆满后，都会触发 GC，为了方便回收，因此就需要将其封装成 oop。这样说，小伙伴们都懂了吗？总之，原来的说法里，对 Java 类的描述模型里有一个 oop 的概念，存储 Java 类实例数据的类也都是以 oop 结尾，为了方便 GC 回收而使用的封装策略也叫作 oop，多种概念混在一起，安能不晕？

5.2.6 oop、klass、handle 的相互转换

oop、klass 和 handle 三者之间是可以相互转换的。

1. 从 oop 和 klass 到 handle

handle 主要用于封装 oop 和 klass，因此往往在声明 handle 类实例的时候，直接将 oop 或者 klass 传递进去，便完成了这种封装。同时，当 JVM 执行 Java 类的方法时，最终也是通过 handle

拿到对应的 oop 和 klass。而为了支持高效快速的调用，JVM 重载了类方法访问操作符->。

oop 类型所对应的 handle 的基类是 Handle，Handle 有如下几种构造函数：

清单：/src/share/vm/runtime/handles.inline.hpp

功能：Handle 类的构造函数

```
inline Handle::Handle(oop obj) {
    if (obj == NULL) {
        _handle = NULL;
    } else {
        _handle = Thread::current()->handle_area()->allocate_handle(obj);
    }
}
```

这个构造函数接受 oop 类型的入参，并将其保存到当前线程在堆区所申请的 handleArea 表中。注意，由于 oop 仅仅是一种指针，因此表中存储的实际上也是指针。

除了这种带有入参的构造函数，还有默认构造函数：

清单：/src/share/vm/runtime/handles.hpp

功能：Handle 类的构造函数

```
Handle() { _handle = NULL; }
```

oop 和 klass 被 handle 封装之后，JVM 内部大部分对 oop 和 klass 的函数调用都要经过 Handle 类。而从 Handle 类到 oop 或者 klass，都必须经过至少一次中间过渡性的寻址，因此为了减少寻址次数，Handle 重载了操作符->：

清单：/src/share/vm/runtime/handles.hpp

功能：Handle 类的构造函数

```
oop non_null_obj() const {
    assert(_handle != NULL, "resolving NULL handle");
    return *_handle;
}

oop operator -> () const {
    return non_null_obj();
}
```

这里定义了 oop operate ->() 操作符重载函数，返回 non_null_obj()，而后者则直接返回 oop 类型的 *_handle 指针，因此如果 JVM 想要调用 oop 的某个函数，可以直接通过 handle。例如，在常量池类型 constantPoolOopDesc 中定义了 void field_at_put(int which, int class_index, int name_and_type_index) 函数，该函数将解析出的常量池元素保存进 constantPoolOop 对象头后面的数据实例区中，在解析常量池的过程中，JVM 并没有直接通过 constantPoolOop 指针来调用

这个函数，而是通过 `constantPoolHandle`。

清单: `/src/share/vm/classfile/classFileParser.cpp`

功能: 通过 `handle` 调用 `oop` 函数举例

```
void ClassFileParser::parse_constant_pool_entries(constantPoolHandle
cp, int length, TRAPS) {

    // 这里省略若干代码

    for (int index = 1; index < length; index++) {
        u1 tag = cfs->get_u1_fast();
        switch (tag) {
            //...
            case JVM_CONSTANT_Fieldref :
                {
                    cfs->guarantee_more(5, CHECK); // class_index,
name_and_type_index, tag/access_flags
                    u2 class_index = cfs->get_u2_fast();
                    u2 name_and_type_index = cfs->get_u2_fast();
                    cp->field_at_put(index, class_index, name_and_type_index);
                }
                break;
            //...
        }
    }
}
```

这里面执行了 `cp->field_at_put()` 函数，而 `cp` 则是 `constantPoolHandle` 类型。

`klass` 体系的 `Handle` 类全都继承于 `KlassHandle` 这个基类，与 `oop` 体系类似，`KlassHandle` 中也定义了多种构造函数用来实现对 `klass` 或 `oop` 的封装，并且重载了 `operate ->()` 函数实现从 `handle` 直接调用 `klass` 的函数。JVM 创建 Java 类所对应的类模型时便使用了这种方式：

清单: `/src/share/vm/classfile/classFileParser.cpp`

功能: 通过 `handle` 调用 `oop` 函数举例

```
instanceKlassHandle ClassFileParser::parseClassFile(Symbol* name,
                                                    Handle class_loader,
                                                    Handle
protection_domain,
                                                    KlassHandle host_class,
                                                    GrowableArray<Handle>*
cp_patches,
                                                    TempNewSymbol&
parsed_name,
                                                    bool verify,
                                                    TRAPS) {
```

```

// 这里省略若干代码

classOop ik = oopFactory::new_instanceClass(name, vtable_size,
itable_size,
static_field_size,
total_oop_map_count,
rt, CHECK_(nullHandle));

instanceClassHandle this_class (THREAD, ik);

// Fill in information already parsed
this_class->set_access_flags(access_flags);
this_class->set_should_verify_class(verify);
jint lh = Klass::instance_layout_helper(instance_size, false);
this_class->set_layout_helper(lh);
assert(this_class->oop_is_instance(), "layout is correct");
assert(this_class->size_helper() == instance_size, "correct
size_helper");
// Not yet: supers are done below to support the new subtype-checking
fields
//this_class->set_super(super_class());
this_class->set_class_loader(class_loader());
this_class->set_nonstatic_field_size(nonstatic_field_size);
this_class->set_has_nonstatic_fields(has_nonstatic_fields);

//...
}

```

在该函数中, 先通过 `classOop ik = oopFactory::new_instanceClass()` 创建了一个 `classOopDesc` 实例, 接着通过 `instanceClassHandle this_class (THREAD, ik)` 将 `oop` 封装到了 `instanceClassHandle` 中, 接下来便通过 `this_class` 指针来直接调用 `instanceClass` 中的各种函数。

2. class 与 oop 的相互转换

为了便于 GC 回收, 每一种 `class` 实例最终都要被封装成对应的 `oop`, 具体操作时, 先分配对应的 `oop` 实例, 接着将 `class` 实例分配到 `oop` 对象头的后面, 从而实现 `oop+class` 这种内存布局结构。对于任何一种给定的 `oop` 和其对应的 `class`, `oop` 对象首地址到其对应的 `class` 对象首地址的距离都是固定的, 因此只要得到了 `oop` 对象首地址, 便能通过偏移固定的距离得到 `class` 对象的首地址。反之, 得到 `class` 对象的首地址后, 也能通过偏移固定的距离得到 `oop` 对象的首地址。通过内存偏移, 便能实现 `oop` 和 `class` 的相互转换。对于每一种 `oop`, 都提供了 `class_part()` 这样的函数, 通过本函数可以直接由 `oop` 得到对应的 `class` 实例。例如 `classOop` 便提供了这种函数:

清单: /src/share/vm/oops/classOop.hpp

功能: oop 转换为 klass

```
Klass* klass_part() const {
    return (Klass*)((address)this + klass_part_offset_in_bytes());
}
```

由于 klass 在内存上相对于 oop 实例位于高地址方向, 因此从 oop 转换到 klass 只需要增加 oop 的首地址。同理, 如果将 klass 转换为 oop, 则只需要对 klass 首地址做减法。例如:

清单: /src/share/vm/oops/class.hpp

功能: klass 转换为 oop

```
klassOop as_klassOop() const {
    return (klassOop) (((char*) this) - sizeof(klassOopDesc));
}
```

下面还是以前文所举的学生类 Student 为例, 假设在某个 Java 方法中连续实例化了 3 个 Student 类, 那么最终在 JVM 内存中将会出现如图 5.12 所示这种布局。

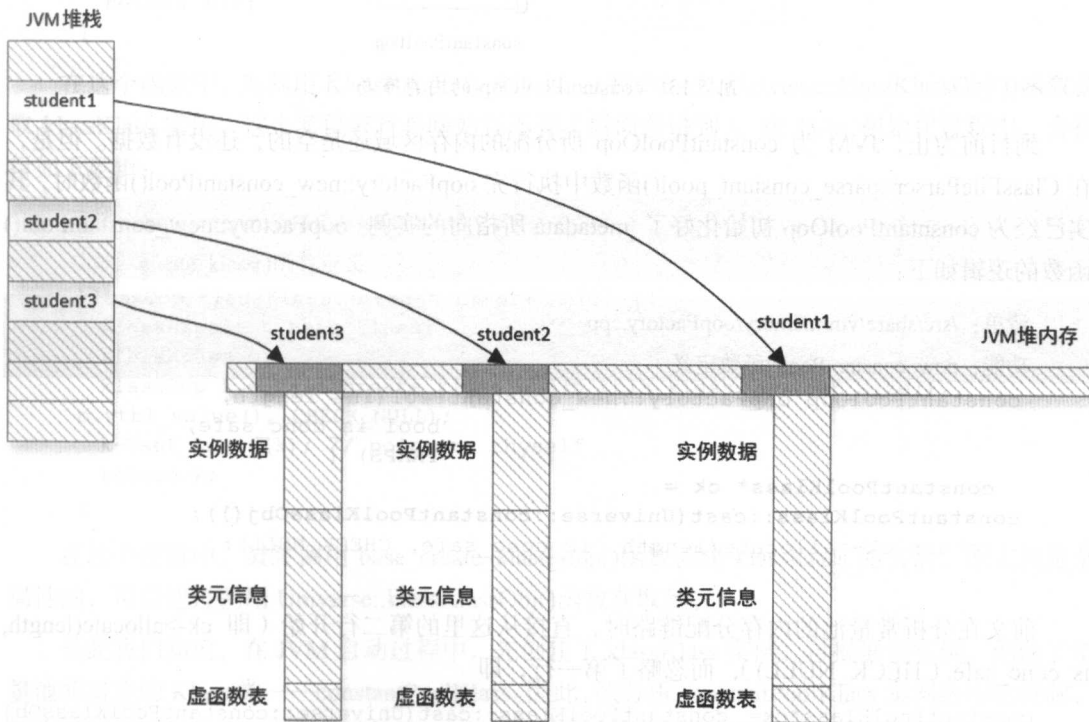


图 5.12 以 Student 类演示 Java 类内存布局

5.3 常量池 klass 模型 (1)

既然在 JVM 内部，每一个对象都会表示为 oop-klass 这种一分为二的模型，那么常量池也不例外。在前面的讲解中，已经分析了 JVM 为 constantPoolOop 所分配的内存大小和内存布局，具体的内存布局如图 5.13 所示。

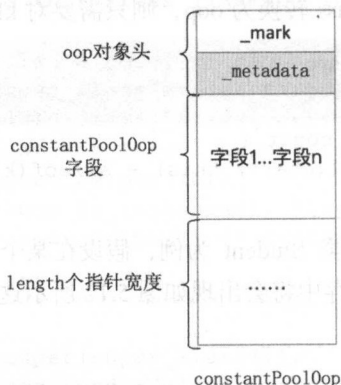


图 5.13 constantPoolOop 的内存布局

到目前为止，JVM 为 constantPoolOop 所分配的内存区域还是空的，还没有数据。但是，在 ClassFileParser::parse_constant_pool() 函数中执行完 oopFactory::new_constantPool() 函数时，其实已经为 constantPoolOop 初始化好了 _metadata 所指向的实例。oopFactory::new_constantPool() 函数的逻辑如下：

清单：/src/share/vm/memory/ooFactory.cpp

功能：new_constantPool() 函数定义

```
constantPoolOop oopFactory::new_constantPool(int length,
                                              bool is_conc_safe,
                                              TRAPS) {

    constantPoolClass* ck =
    constantPoolClass::cast(Universe::constantPoolClassObj());
    return ck->allocate(length, is_conc_safe, CHECK_NULL);
}
```

前文在分析常量池的内存分配链路时，直接从这里的第二行开始（即 ck->allocate(length, is_conc_safe, CHECK_NULL)），而忽略了第一行，即

```
constantPoolClass*ck= constantPoolClass::cast(Universe::constantPoolClassObj
());
```


这行代码调用全局对象 Universe 的静态函数 `constantPoolKlassObj()` 来获取 `constantPoolKlass` 实例指针, `Universe::constantPoolKlassObj()` 函数逻辑如下:

```
static klassOop constantPoolKlassObj() {
    return _constantPoolKlassObj;
}
```

这个函数直接返回了全局静态变量 `_constantPoolKlassObj`。该变量在 JVM 启动过程中被实例化, 在 JVM 初始化过程中, 会调用如下逻辑:

清单: `/src/share/vm/oops/constantPoolKlass.cpp`

功能: `create_klass()` 函数定义

```
klassOop constantPoolKlass::create_klass(TRAPS) {
    constantPoolKlass o;
    KlassHandle h_this_klass(THREAD, Universe::klassKlassObj());
    KlassHandle k = base_create_klass(h_this_klass, header_size(),
    o.vtbl_value(), CHECK_NULL);
    java_lang_Class::create_mirror(k, CHECK_NULL); // Allocate mirror
    return k();
}
```

在这个函数中, 先调用 `KlassHandle h_this_klass(THREAD, Universe::klassKlassObj())` 函数获取 `klassKlass` 实例。这个实例究竟是啥姑且不管 (后面会讲到), 在 JVM 初始化过程中, 会执行如下逻辑:

清单: `/src/share/vm/oops/klassKlass.cpp`

功能: `create_klass()` 函数定义

```
klassOop klassKlass::create_klass(TRAPS) {
    KlassHandle h_this_klass;
    klassKlass o;
    klassOop k = base_create_klass_oop(h_this_klass, header_size(),
    o.vtbl_value(), CHECK_NULL);
    k->set_klass(k); // point to thyself
    return k;
}
```

在这个逻辑中, 最终调用 `base_create_klass_oop()` 函数创建 `klassKlass` 的实例, 该实例是全局性的, 可以通过调用 `Universe::klassKlassObj()` 函数获取到。

至此我们知道, 在 JVM 启动过程中, 先创建了 `klassKlass` 实例, 再根据该实例, 创建了常量池所对应的 `Klass` 类——`constantPoolKlass`。因此, 欲分析 `constantPoolKlass` 实例的构建机制, 首先就要分析 `klassKlass` 实例的构建。下面就先从 `klassKlass` 实例的构建原理说起。

5.3.1 classKlass 实例构建总链路

先上一张 classKlass 实例构建的总体链路图，如图 5.14 所示。

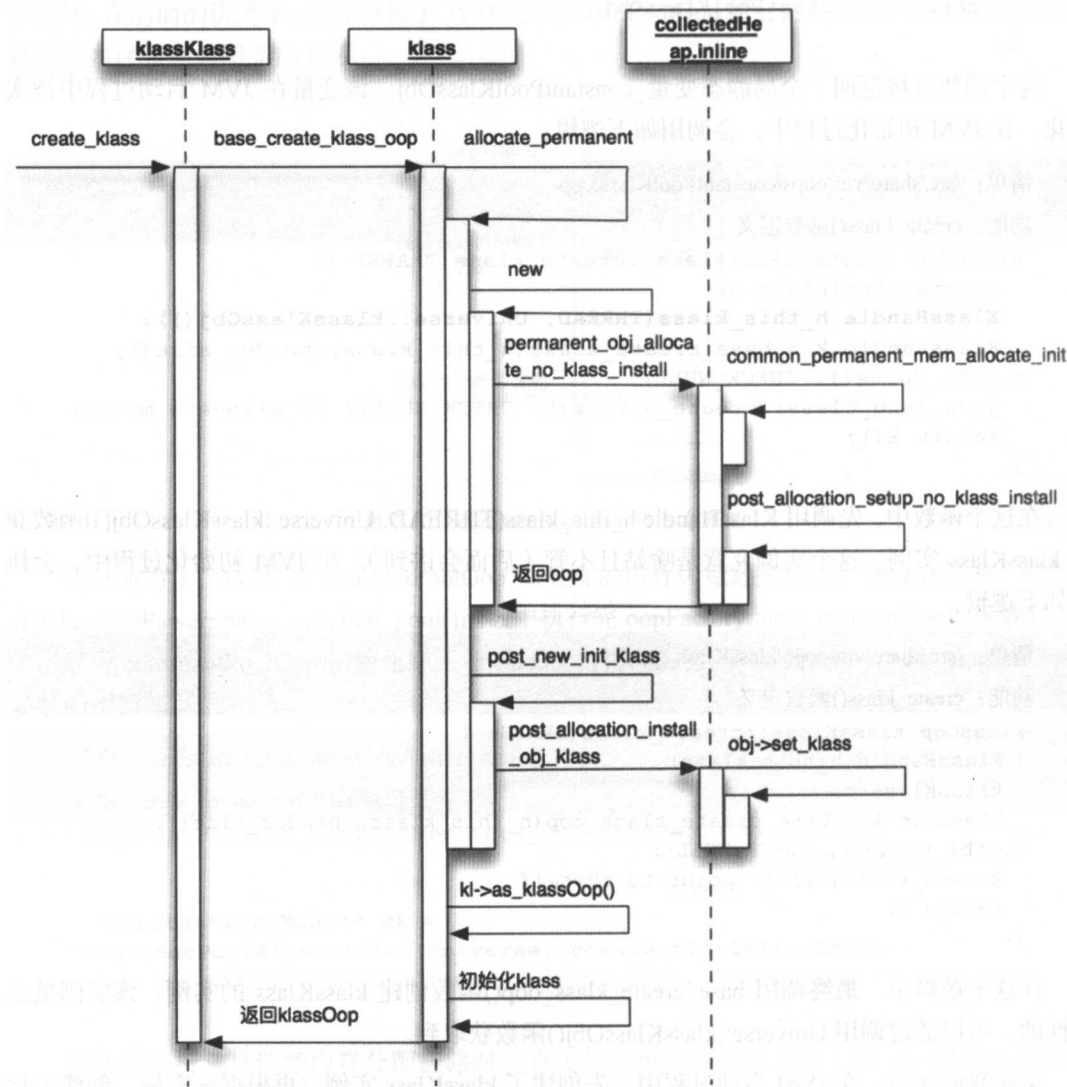


图 5.14 classKlass 实例构建的总体链路

classKlass.cpp::create_class()函数位于非常核心的位置,该位置也是理解 JVM 内存模型的关键,将这个理解透了,JVM 内存模型便明白了一大半。由于这个函数实在太重要了,因此十分

有必要对其进行详细讲解。

图 5.14 所示的调用链路虽然相比于 JVM 内部其他链路已经显得很简单了,但是这些方法分布在不同的源代码文件中,阅读时需要在不同的源代码文件中反复切换,因此下面通过将函数展开并将源码贴在一起(大家千万不要以为这是在浪费纸张,整本书里并没有几处会大肆粘贴源代码来占篇幅,实在是这里的代码太核心了),让大家更容易理清头绪:

```
//-----begin classKlass.cpp create_class()-----
KlassHandle h_this_klass;
KlassKlass o;
KlassOop k = base_create_class_oop(h_this_klass, header_size(), o.vtbl_value(), CHECK_NULL);

//-----begin klass.cpp base_create_class_oop()-----
size = align_object_size(size);
Klass* kl = (Klass*) vtbl.allocate_permanent(klass, size, CHECK_NULL);

//-----begin klass.hpp allocate_permanent()-----
void* result = new(klass_klass, size, THREAD) thisKlass();

//-----begin klass.hpp allocate_permanent()-----
KlassOop k =
    (KlassOop)
    CollectedHeap::permanent_obj_allocate_no_klass_install(klass, size, CHECK_NULL);

//-----begin collectedHeap.inline.hpp
permanent_obj_allocate_no_klass_install()--
HeapWord* obj = common_permanent_mem_allocate_init(size, CHECK_NULL);

//-----begin collectedHeap.inline.hpp
common_permanent_mem_allocate_init()--
// ① 为 KlassOop 申请内存
HeapWord* obj = common_permanent_mem_allocate_noinit(size,
CHECK_NULL);

// ② KlassOop 内存清零
init_obj(obj, size);
return obj;
//-----end collectedHeap.inline.hpp
common_permanent_mem_allocate_init()----

// ③ 初始化 KlassOop._mark 标识
post_allocation_setup_no_klass_install(klass, obj, size);

//---begin collectedHeap.inline.hpp
post_allocation_setup_no_klass_install()
```

```

        oop obj = (oop)objPtr;
        if (UseBiasedLocking && (klass() != NULL)) {
            obj->set_mark(klass->prototype_header());
        } else {
            obj->set_mark(markOopDesc::prototype());
        }
        //---end collectedHeap.inline.hpp
post_allocation_setup_no_class_install()

        return (oop)obj;
        //-----end collectedHeap.inline.hpp
permanent_obj_allocate_no_class_install()----

        return k->class_part();
        //-----end klass.hpp allocate_permanent()-----

        if (HAS_PENDING_EXCEPTION) return NULL;
        klassOop new_klass = ((Klass*) result)->as_klassOop();
        OrderAccess::storestore();
        // ④ 初始化 klassOop._metadata
        post_new_init_klass(klass_klass, new_klass, size);
        return result;
        //-----end klass.hpp allocate_permanent()-----

klassOop k = kl->as_klassOop();

// 这里省略部分逻辑

// ⑤ 初始化 klass
kl->set_java_mirror(NULL);
kl->set_modifier_flags(0);
kl->set_layout_helper(Klass::_lh_neutral_value);
kl->set_name(NULL);
AccessFlags af;
af.set_flags(0);
kl->set_access_flags(af);
kl->set_subklass(NULL);
kl->set_next_sibling(NULL);
kl->set_alloc_count(0);
kl->set_alloc_size(0);

kl->set_prototype_header(markOopDesc::prototype());
kl->set_biased_lock_revocation_count(0);
kl->set_last_biased_lock_bulk_revocation_time(0);

```

```
return k;
//-----end class.cpp base_create_class_oop()-----
```

```
// ⑥ 自指
k->set_class(k); // point to thyself
return k;
//-----end classClass.cpp create_class()-----
```

上面就是整合之后的源代码,其实代码量并不是很大(当然里面还是省略了部分函数展开)。

这部分代码逻辑大体上可以划分为 6 个步骤,在上面的这段整合好的代码中分别使用①、②、③等这样的编号进行标注,这 6 个步骤如图 5.15 所示。



图 5.15 classOop 实例化的 6 个步骤

下面将从这 6 个方面按照程序主线依次讲解。

5.3.2 为 classOop 申请内存

从整合之后的源代码可以看出,从进入 `classClass.cpp::create_class()` 之后,就开始一路调用被层层封装起来的函数,除了函数调用之外并没有其他复杂的逻辑,一直调用到 `CollectedHeap::permanent_obj_allocate_no_class_install()`,才终于消停,开始做正事了。函数名

不小心暴露了这件正事，顾名思义，这里开始在永久区为 obj 对象分配内存了。这个函数的实现逻辑是：

清单：/src/share/vm/gc_interface/collectedHeap.inline.hpp

作用：permanent_obj_allocate_no_class_install()函数

//为 oop 申请内存

```
HeapWord* obj = common_permanent_mem_allocate_init(size, CHECK_NULL);
```

//初始化 oop 标识

```
post_allocation_setup_no_class_install(klass, obj, size);
```

这个函数主要干了两件正事：内存申请和标识初始化。申请内存调用的是 `common_permanent_mem_allocate_init()` 函数，这个函数在前文讲解 `ConstantPoolOop` 的内存初始化时详细讲解过，其主要功能就是根据传入的 `size` 在永久区划分出一块指定大小的内存区域。函数的实现不再赘述，但是 `size` 的大小需要关注。这里的 `size` 参数值的源头在 `klassKlass.cpp::create_klass()` 函数中，该函数调用 `base_create_klass_oop(h_this_klass, header_size(), o.vtbl_value(), CHECK_NULL)` 时，第 2 个参数是 `header_size()` 函数所返回的值，要知道 `size` 参数的值，只需要看看 `header_size()` 函数的实现即可。该函数定义在 `klassKlass.hpp` 文件中，定义如下：

```
static int header_size() {
    return oopDesc::header_size() + sizeof(klassKlass)/HeapWordSize;
}
```

这个定义表明，`size` 的构成包含 2 部分：`oopDesc` 和 `klassKlass` 的类型所占内存空间。这两个类型本身的大小在编译期间便可知道。

为什么会是这么大呢？别忘了本节的主题——实例化 `klassKlassOop`。既然要实例化 `klassKlassOop`，就得为其分配足够的内存空间，而前文讲过，JVM 内部通过“两模型三维度”去描述一个对象，两个模型自然就是 `oop` 和 `klass`。`klassKlass` 这个类实例在 JVM 内部也是被描述的对象，因此 JVM 也将这个对象模型一分为二，分别使用 `oop` 和 `klass` 这两个拆分的模型来描述。

当 `common_permanent_mem_allocate_init()` 函数执行完之后，JVM 的永久区中便多了一个对象内存布局，该对象是 `klassKlassOop`，其内存布局模型如图 5.16 所示。



图 5.16 classKlass 内存布局模型

图 5.16 中的 classKlass 实例区其实就是 classKlass 的实例对象所在的地方，该区域中的实例数据都是 classKlass 类的成员变量。

虽然直到这里，并没有明确提出这块内存区域就隶属 classKlassOop 这个对象，但是并不妨碍我们大胆猜测，因为 JVM 既然将 classKlass 也看作是一个对象，那么 JVM 一定会将其包装成一个 oop，这样才方便垃圾统一回收。后面会通过 JVM 的源码来进行证实。

但是等等，等等，等一下，这里貌似有点不一样，好像不是那么回事，似乎哪里不一致。具体是哪里呢？你稍等，让我理下思路先。

前文讲过 JVM 的常量池 constantPoolOop 的内存分配，对于常量池对象，对于 constantPoolOop 对象，JVM 的内存模型布局如图 5.17 所示。

对比下 classKlassOop 和 constantPoolOop 这两个对象的内存布局，其对象头都是一致的，但是紧跟在对象头后面的数据区的数据则大不相同。constantPoolOop 的数据区由 2 部分组成：constantPoolOop 自己的字段和长度等同于一个 Java 类编译后得到的常量池所有元素所占内存区域。而到了 classKlassOop 对象，其数据区则直接变成了 classKlass 类型实例。这里有一个点疑问：classKlassOop 的对象头后面直接跟了 class 模型实例，class 模型实例一般用于描述对象的元数据，但是 oop 本身含有一个指针 _metadata 指向这个元数据区域，那么 oop 的 _metadata 指针指向哪里，还有用吗？或者难道是直接指向其自身？

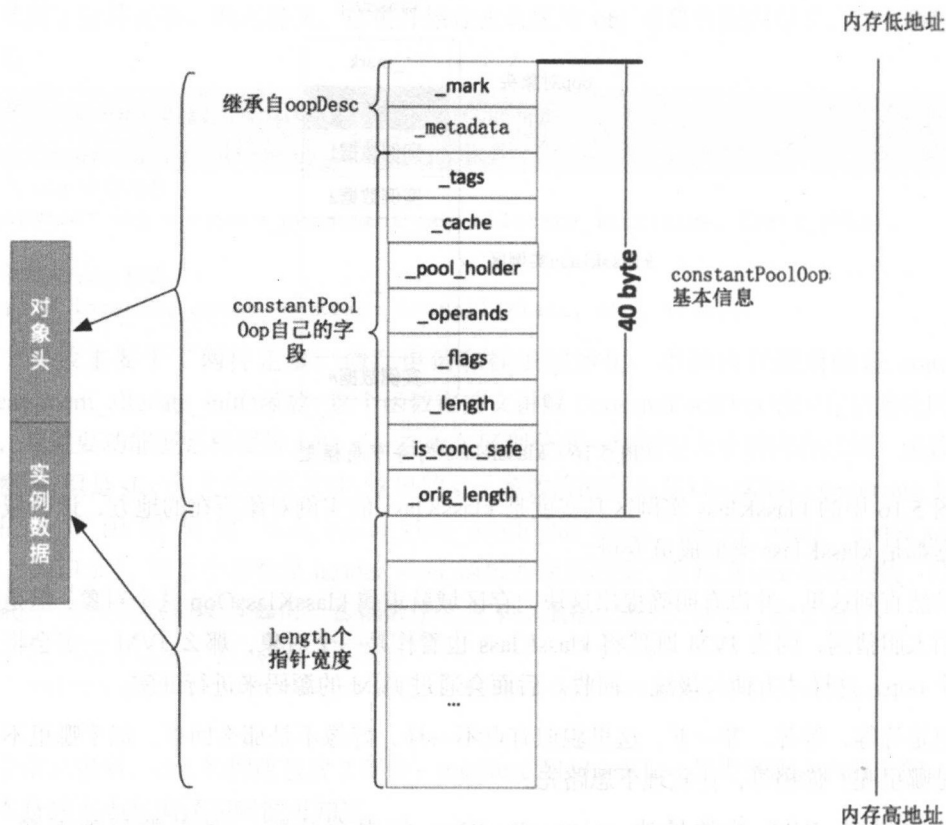


图 5.17 constantPoolOop 对象内存布局模型

如果你还不明白，可以这样思考，在 JVM 内部按照两模型三维度去描述一个对象，那么如果按照这种规范来描述 `klassKlassOop`，则 `klassKlassOop` 的内存布局应该是如图 5.18 所示的样子。

在图 5.18 中，`klassKlassOop._metadata` 指向了一个专门的元数据区，这样 Java 类通过对类实例进行反射便能在运行期动态获取到类型的结构信息。但是很明显，真实的情况是，`klassKlassOop` 的类元信息直接跟在了 `oop` 对象头后面，`oop` 后面跟的并不是 `oop` 本身的实例数据。既然如此，那么 `_metadata` 指针最终会指向哪里呢？要想知道答案，还得继续往下看代码。

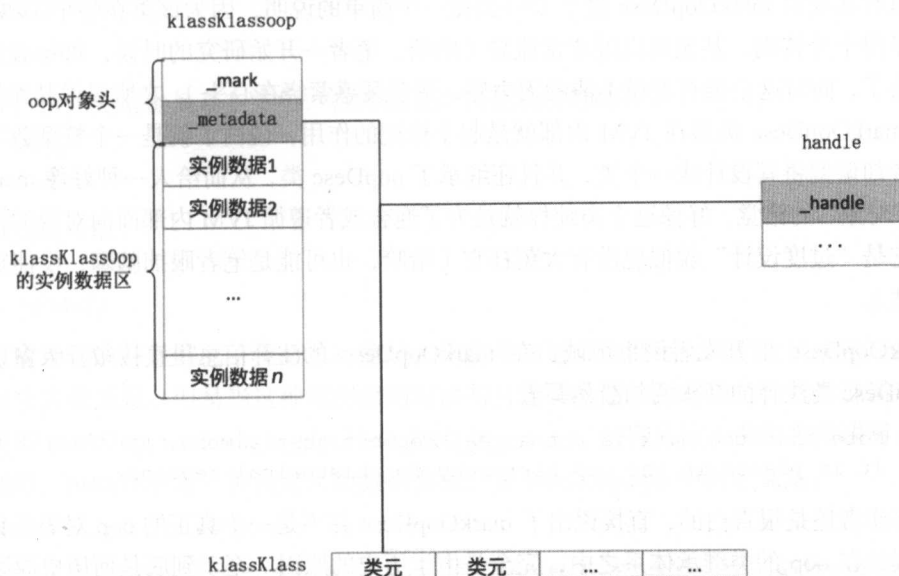


图 5.18 使用“三维度”模型描述 constantPoolOop

5.3.3 klassOop 内存清零

为 klassOop 分配完内存后，接下来开始将这段内存清零。清零调用函数 `collectedHeap.init_obj()`，该函数的实现思路在前文讲解 `constantPoolOop` 的内存清零时已经讲过。之所以要清零，是因为 JVM 的永久区是一个可重复使用的内存区域，会被 GC 反复回收，因此 JVM 为 oop 刚申请的内存区域里可能还保存着原来已经被清理的对象的数据。

5.3.4 初始化 mark

通过前面两个步骤，已经成功地为 klassOop 申请了内存并实现清零，那么接下来的逻辑自然是往内存里填充数据。oop 类里面一共包含两个成员变量：`_mark` 和 `metadata`，这里先填充 `_mark` 变量。`_mark` 变量的填充主要通过调用 `post_allocation_setup_no_klass_install()` 函数实现。在上面整合的源码中贴出了该函数的实现，其中主要调用了 `obj->set_mark(markOopDesc::prototype())` 函数。

这里有必要对 `markOopDesc` 这个 C++ 类做一个简单的说明，因为该类在整个 JVM 的源代码中都显得十分特别，甚至可以说非常诡异（哈哈，笔者一开始研究的时候，每每看到这里就读不下去了，面对这个类有着说不清的无力感，诸多疑惑萦绕在心头）。之所以说其诡异，是因为其实 `markOopDesc` 类型在 JVM 内部就是起个标记的作用，说白了就是一个整型数字，但是其设计者却偏偏将其设计成一个类，并且还继承了 `oopDesc` 类，从而给人一种好像 `markOop` 也是一个“对象”的错觉，好像这个类纯粹就是为了迎合或者遵循 JVM 内部面向对象的设计风格，说它是“过度设计”貌似也没有太冤枉它（哈哈，也可能是笔者眼拙脑笨，没能窥透这里面的道理）。

`markOopDesc` 的开发者倒也坦诚，在 `markOopDesc` 的注释信息里直接敞开心窗说亮话，`markOopDesc` 类注释的开头两句赫然写着：

```
Note that the mark is not a real oop but just a word.
It is placed in the oop hierarchy for historical reasons.
```

看来作者还是很直白的，直接说出了 `markOopDesc` 并不是一个真正的 `oop` 对象指针，所以也将其放在 `oop` 的类继承体系之中，完全是由于历史的原因。至于到底是何历史原因，个人并没有继续深究，大家有兴趣可以去找这方面的资料。

总之，这个类非常容易使人困惑，如果顺着 `oop` 这套模型去分析，反而会走上“邪道”，并且最终会走进死胡同。

`markOopDesc` 类除了从 `oopDesc` 类继承而来的两个成员变量，并没有自己的成员变量，只是里面定义了不少枚举。这些枚举类型本身并不占用内存空间。继续上面的话题，在为 `klassOop` 填充 `_mark` 成员变量时，调用了 `obj->set_mark(markOopDesc::prototype())` 函数，其入参是 `markOopDesc::prototype()`，该函数声明如下：

清单：/src/share/vm/oops/markOopDesc.hpp

作用：prototype() 函数

```
static markOop prototype() {
    return markOop( no_hash_in_place | no_lock_in_place );
}
```

在 `prototype()` 函数里貌似返回了 `markOop` 的构造函数，这个构造函数的入参是一个整型变量。先从 `markOopDesc.hpp` 文件中搜索这样的构造函数，结果很遗憾搜不到。由于 `markOopDesc` 继承了 `oopDesc` 类，但是 `oopDesc` 类中也没有定义任何一个类似的构造函数。事实上，这里并不是调用了 `markOop` 的构造函数，而是 C 语言的内建类型直接赋初值的写法。在 C 语言中，对于变量初始化，有一种快速赋初值的写法，例如：

```
int x=3;
```

可以直接写成:

```
int x(3);
```

同样, 指针类型也是基本类型, 自然也支持快速赋初值的写法, 例如:

```
int x=3;
int *p=&x;
```

可以直接写成:

```
int x=3;
int *p(&x);
```

本来这种写法一般情况下是简单易懂的, 即使不知道有这种写法的人看了这种写法之后, 也能猜出个大概意思。但是当这种赋初值的写法与自定义类型结合起来时, 就容易误导人了, 例如这里的 `markOop(no_hash_in_place|no_lock_in_place)`, 就容易使人误以为在调用 `markOop` 的构造函数。`markOop` 是一种自定义的数据类型, 在 `hierarchy.hpp` 中的定义是:

```
typedef class markOopDesc* markOop;
```

由此可见, `markOop` 的实际类型是 `markOopDesc*` 指针类型, 而指针是一种基本的数据类型, 因此自然支持赋初值的写法。下面举例说明:

清单: test.cpp

作用: 演示指针赋初值

```
#include <stdio.h>
```

```
class A{
public:
    enum{
        a=1,
        b=2
    };
};
```

```
//自定义一种数据类型 AT
typedef class A* AT;
```

```
AT test(){
    //为指针赋初值
    return AT(3);
}
```

```
int main(){
    printf("yy=%p\n", test());
}
```

本例中,先定义类型 A,接着声明一种自定义的数据类型 AT,AT 的类型原型是 A*,其实是一种指针。在 test()函数中,采用赋初值的写法直接返回类型 AT 的变量。最终打印出来的结果是 3,因为 test()返回了一个指针,而指针里所存储的值就是 3。

其实 C++编译器在解释 test()函数时,会将语法最终展开为下面这种形式:

```
A* test() {
    //为指针赋初值
    A* a=(A*) 3;
    return a;
}
```

这里需要注意的是,由于指针类型变量本质上存储的也是整型数据(因为内存地址一定是整数),因此可以将一个整数直接强制转换成一个指针类型。

基于这里的例子,回头看 markOop::prototype()函数,一样的道理,该函数最终会在编译期展开为下面这种逻辑:

```
static markOopDesc* prototype() {
    markOopDesc* mark=(markOopDesc*)(no_hash_in_place | no_lock_in_place );
    return mark;
}
```

这样一来,prototype()函数的逻辑就再清晰不过了,该函数最终将返回一个指针,这个指针里存储了一个整数数字。由于 oop._mark 成员变量用于存储 JVM 内部对象的哈希值、线程锁等信息,而 prototype()函数所返回的 mark 标识则标记这个 oop 当前尚未建立唯一哈希值,并且也没有加任何锁。哈希值相对于 JVM,就类似于人的身份证之于社会,此时的这个 oop 对象就像一个刚出生的婴儿一样,尚依偎在母亲“JVM”的襁褓中,甚至在这个世界上连唯一的身份证都没有。

JVM 内部每次读取 oop 的 mark 标识时,会调用 markOopDesc 的 value()函数,该函数定义如下:

```
uintptr_t value() const {
    return (uintptr_t) this;
}
```

value()函数直接返回 mark 指针的值,由此可见,markOopDesc 内部其实是将 this 指针当作它的值来使用的,而并没有将 markOop 指针还原成 markOopDesc 对象来使用。所以,markOop 虽然看起来是指针但其实并不是真的指向 markOopDesc 实例的指针,这正是 markOopDesc 这个类型神奇的地方。

如果你认为 oop 的 mark 标识是一个指向 markOopDesc 实例的指针,那么在分析 klassOop 时,你可能会认为 JVM 执行完 markOopDesc::prototype()函数,初始化完 klassOop._mark 成员变

量之后的内存布局如图 5.19 所示。

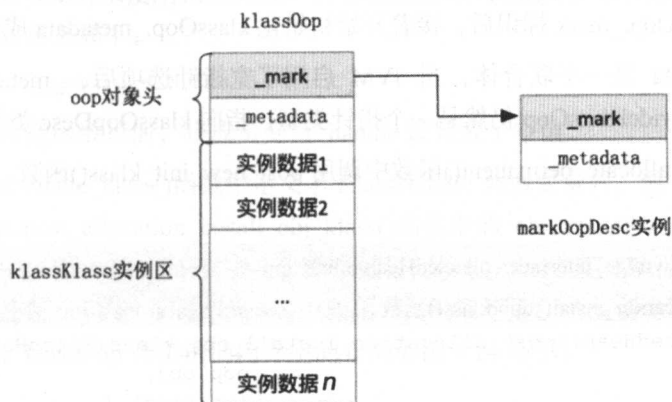


图 5.19 一种假设的内存布局图

如果真是这样就陷入了死胡同，因为 JVM 内部根本就没有在任何地方构建 markOopDesc 实例对象。

正是由于 markOop 指针其实并不是一个真正的 oopDesc 类型，仅仅是一个指针，并且其作用仅仅用于存储 JVM 内部对象的哈希值、锁状态标识等信息，因此 JVM 执行完 markOopDesc::prototype() 函数之后，正确的内存布局应该如图 5.20 所示。

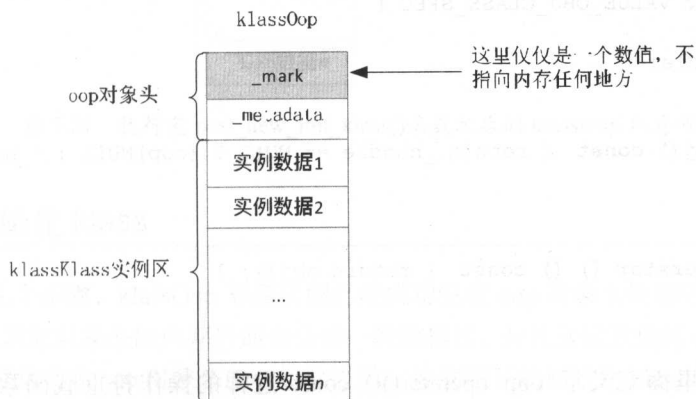


图 5.20 mark 字段的含义

5.3.5 初始化 klassOop._metadata

初始化完 klassOop._mark 标识后，接着开始初始化 klassOop._metadata 成员。

oop 的 _metadata 是一个联合体，当 JVM 启用了宽指针选项后，_metadata 的类型便是 wideKlassOop，而 wideKlassOop 仍然是一个指针类型，指向 klassOopDesc 类型实例。

在 klass.hpp 的 allocate_permanent() 函数中调用 post_new_init_klass() 函数，而该函数最终调用的方法如下：

清单：/src/share/vm/gc_interface/collectedHeap.inline.hpp

作用：post_allocation_install_obj_klass() 函数

```
void CollectedHeap::post_allocation_install_obj_klass(KlassHandle klass,
                                                         oop obj,
                                                         int size) {
    obj->set_klass(klass());
}
```

该函数通过调用 obj->set_klass() 来完成 _metadata 成员变量的赋值，该函数的入参是 klass()，这并不是在调用 KlassHandle 的构造函数，因为在 Handle 类中，() 操作符被重载，因此这里的 klass() 其实是在调用普通函数。在 Handle 类中有如下定义：

清单：/src/share/vm/runtime/handles.hpp

作用：post_allocation_install_obj_klass() 函数

```
class Handle VALUE_OBJ_CLASS_SPEC {
private:
    oop* _handle;

protected:
    oop obj() const { return _handle == NULL ? (oop) NULL : *_handle; }
    //...

    oop operator () () const { return obj(); }
    //...
};
```

在 Handle 类里面定义了 oop operator()() const 这样的操作符重载函数，很显然，在 CollectedHeap::post_allocation_install_obj_klass() 函数中调用 obj->set_klass(klass()) 时，入参 klass() 最终调用了 Handle 类的 obj() 函数，而该函数直接返回 Handle 类里面的 _handle 变量。

于是问题变成了分析 _handle 指针到底指向哪里，这需要从源头上分析。将目光重新回到整条链路的源头——klassKlass::create_klass() 函数。CollectedHeap::post_allocation_install_obj_klass()

函数中所调用的 `obj->set_class(class())` 的入参的 `class` 变量便是在该函数中定义的,定义方式是:
`KlassHandle h_this_class`,未使用 `new` 关键字,因此这会调用 `Handle` 类的默认构造函数,而 `Handle` 类的默认构造函数定义如下:

```
Handle() { _handle = NULL; }
```

在这个默认构造函数里面,将成员变量 `_handle` 设置成了空值。

在 `klassKlass::create_class()` 函数中定义了 `KlassHandle` 类型变量后,便一路透传,直到透传到 `CollectedHeap::post_allocation_install_obj_class()` 函数中的 `obj->set_class(class())` 函数的入参,并且在整个过程中, `KlassHandle` 类型变量都未做任何修改,因此其内部的 `_handle` 成员变量一直都是空值,于是在 `obj->set_class(class())` 中调用 `class()` 后所返回的值便也是空值。所以,当这一步执行完之后, `klassOop` 类实例的内存空间布局如图 5.21 所示。

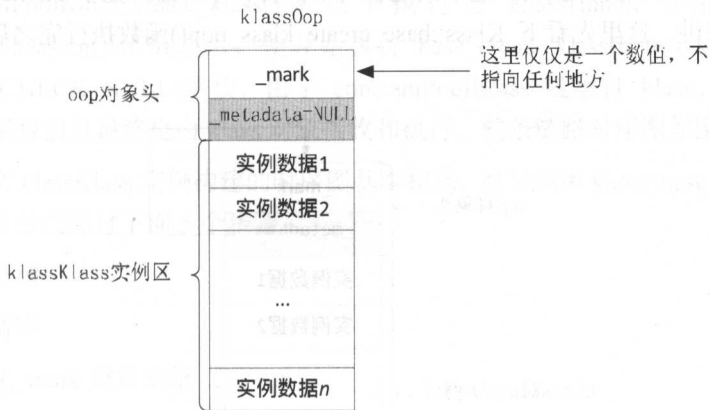


图 5.21 执行完 `post_new_init_class()` 函数之后的 `klassOop` 内存布局

5.3.6 初始化 klass

经过前面几个步骤, `klassOop` 对象实例已经成功完成 `oop` 对象头的初始化。由于 `klassOop` 也是一个 `oop`, 因此对象头的内存后面也会接一段数据区, 并且这段数据区正是 `klassKlass` 类型实例存放地。因此完成 `oop` 对象头的初始化之后, 接着便开始初始化 `klassKlass` 类型实例。

在 `Klass::base_create_class_oop()` 函数中, 执行完 `Klass* kl = (Klass*) vtbl.allocate_permanent(klass, size, CHECK_NULL)` 函数之后, 便得到 `klass` 对象实例的指针, `Klass::base_create_class_oop()` 函数中后续的逻辑便都是在处理 `klass` 对象实例的初始化, 具体代码在上面的整合代码中已经贴出, 可以看到此时大部分属性都被赋为空值了。注意这里也调用

了 `kl->set_prototype_header(markOopDesc::prototype())` 函数来设置对象标, 这说明 JVM 内部虽然将 `klass` 也封装成了 `oop`, 但是 `klass` 毕竟是独立的一种类型, 因此也需要记录线程锁等相关标识。

5.3.7 自指

当执行完 `Klass::base_create_class_oop()` 函数之后, CPU 回到了 `klassKlass::create_klass()` 函数中, 开始执行 `k->set_klass(k)` 这个逻辑。这里的 `k` 是 `klassOop`, 是 `klassOopDesc*` 类型的指针。

`k->set_klass(k)` 其实是在设置 `klassOopDesc` 类型实例的 `_metadata` 成员变量, 虽然在前面的步骤中, `_metadata` 成员变量已经被设置过一回, 但是当时被设置成了 `NULL` 空值, 这一次再次设置, 则是专门为 `klassOop` 量身定做了。在本步骤中, `klassOop` 的 `_metadata` 成员变量被设置为指向其自身。为什么要指向自己呢? 这个问题暂时不回答, 等后面讲完了 `constantPoolOop` 的完整内存布局后再来讲。这里先看下 `Klass::base_create_class_oop()` 函数执行完之后的内存布局, 如图 5.22 所示。

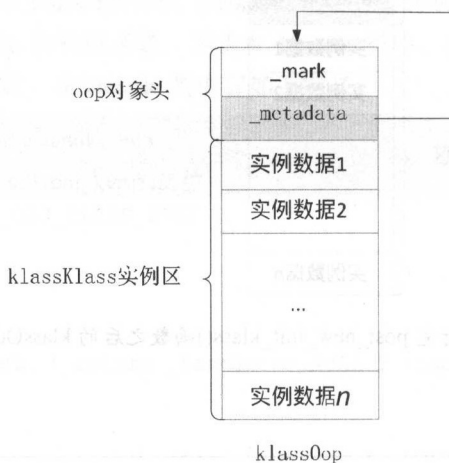


图 5.22 执行完 `Klass::base_create_class_oop()` 函数之后的 `klassOop` 内存布局

至此, `klassOop` 的实例化便完成了, 最终得到的 `klassOop` 的内存模型便是如图 5.22 所示的样子。虽然最终只得到了 `oop`, 但是其实内存里是有两个类型实例的, 一个是 `oop` 对象头实例, 紧接其后的则是 `klass` 类型实例。通过 `oop` 对象头的内存位置可以得到 `klass` 类型实例的首地址, 进行转换后拿到 `klass` 实例。在这整个过程中, 笔者想讲解一下这里面所使用到的一些 C++ 特性技巧。

5.4 常量池 klass 模型 (2)

5.4.1 constantPoolKlass 模型构建

上文分析了 `klassKlass` 对象实例构建的原理，而且提到，之所以要分析 `klassKlass` 的构建，是因为这是研究常量池 `constantPoolKlass` 构建的基础，因为在 JVM 启动过程中，JVM 需要执行 `constantPoolKlass::create_klass()` 函数来构建 `constantPoolKlass` 实例，而在该函数中调用了 `KlassHandle h_this_klass(THREAD, Universe::klassKlassObj())` 函数来获取 `klassKlass` 所对应的 handle，JVM 以此为基础来构建常量池所对应的 `klass`。

在 `constantPoolKlass::create_klass()` 函数中执行完 `KlassHandle h_this_klass(THREAD, Universe::klassKlassObj())` 函数之后，便开始执行 `base_create_klass(h_this_klass, header_size(), o.vtbl_value(), CHECK_NULL)` 函数，由于 `constantPoolKlass` 继承自 `klass`，因此这里的 `base_create_klass()` 函数消息最终也由 `klass` 对象接收和执行。整条链路时序图如图 5.23 所示。

此图与上文 `klassKlass` 实例构建的时序图基本相同，通过调用 `klass::base_create_klass_oop()` 入口进入，之后分别经过下面 5 个步骤：

- (1) 内存申请
- (2) 内存清零
- (3) 初始化 `_mark` 成员变量
- (4) 初始化 `_metadata` 成员变量
- (5) 初始化 `klass`

最终返回包装好的 `klassOop`。

由于 `constantPoolKlass` 实例构建的过程与 `klassKlass` 实例构建的过程完全一致，因此这里不再进行详细讲解，只需要关注最后的结果即可。很显然，JVM 最终在永久区所创建的对象是 `constantPoolKlass`，但是由于每一个 `klass` 最终都会被包装成 `oop`，因此最终在内存中所构建的模型如图 5.24 所示。

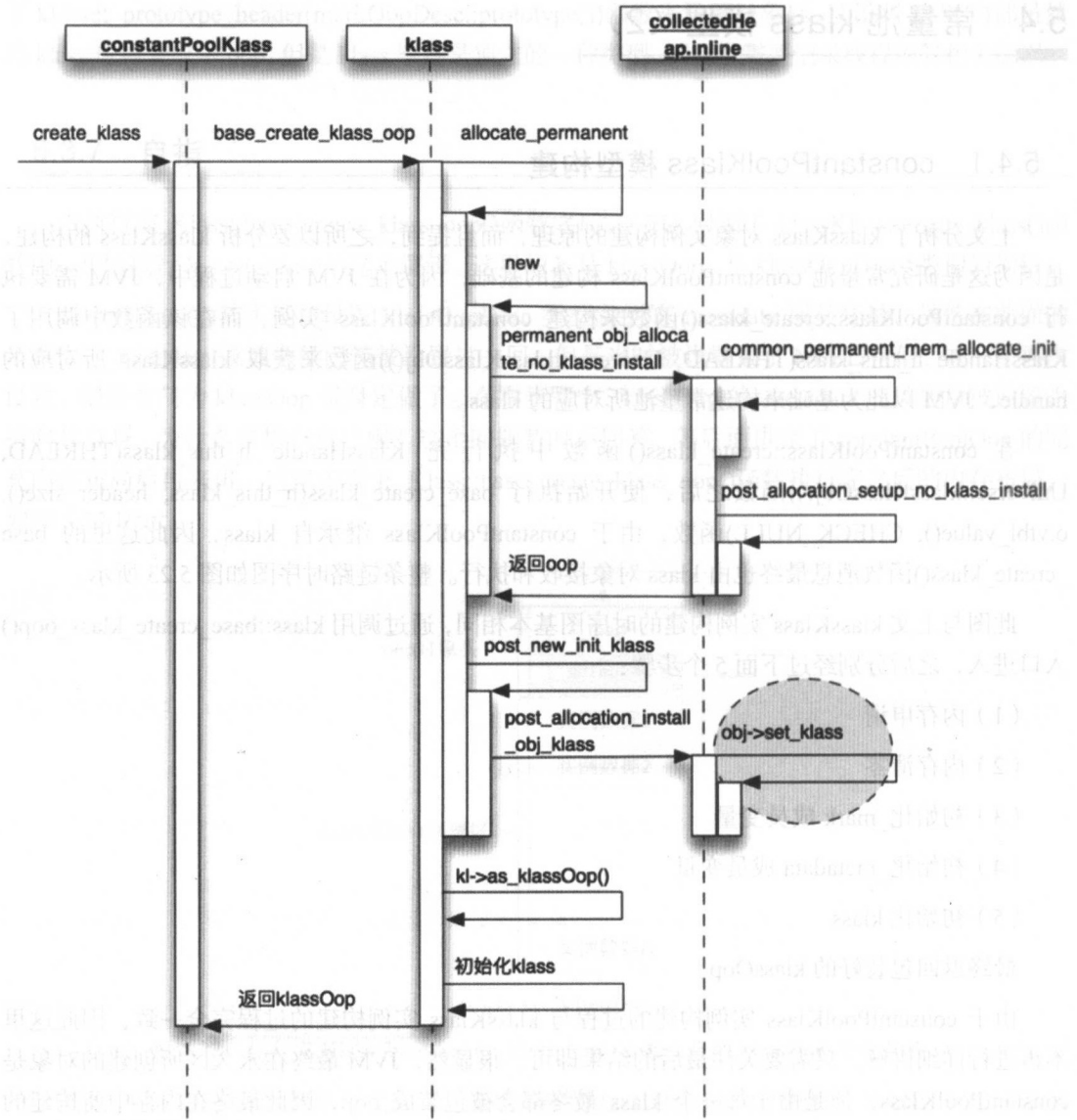


图 5.23 constantPoolKlass 实例构建时序图

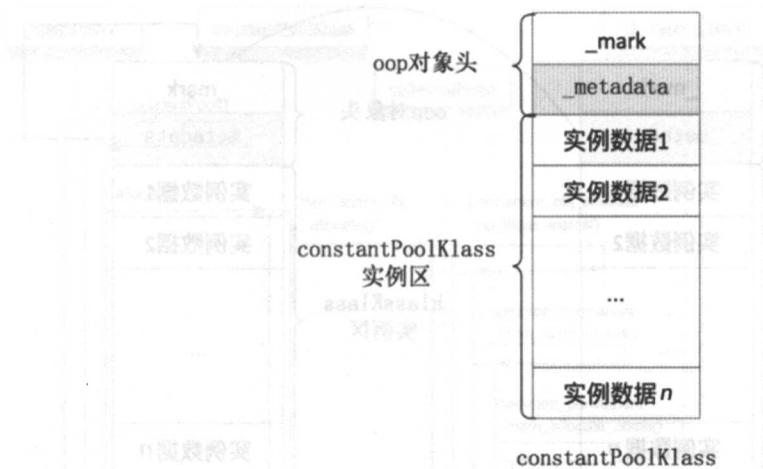


图 5.24 constantPoolKlass 内存布局模型

当然，为了构建这么一个模型，从一开始就要准确计算出其所占内存大小，计算出的大小将通过 `klass::base_create_class_oop()` 函数的 `size` 入参传递给后续流程。在构建 `constantPoolKlass` 时，这个大小由 `constantPoolKlass::header_size()` 函数所确定：

```
static int header_size() {
    return oopDesc::header_size() + sizeof(constantPoolKlass)/HeapWordSize;
}
```

由这个逻辑可知，JVM 在构建 `constantPoolKlass` 时，所分配的内存大小为 `oopDesc` 的大小与 `constantPoolKlass` 的大小之和，即 JVM 将封装 `constantPoolKlass` 的 oop 的大小已经计算进去了。

JVM 构建 `constantPoolKlass` 的逻辑与构建 `klassKlass` 的逻辑基本是一致的，但入参 `size` 不同，且在调用 `klass::base_create_class()` 函数时所传入的 `KlassHandle` 入参也不相同，`KlassHandle` 入参的不同将决定最终所构建出来的 `klassOop` 的 `_metadata` 参数不同。在 `klass::base_create_class()` 入口后面的链路中，有一步调用了 `obj->set_class()`，这一步便是在设置 oop 的 `_metadata` 成员变量。前面分析过，在构建 `klassKlass` 时，所传入的 `KlassHandle` 其实是 `NULL`，因此在 `klassKlass::create_class()` 中又调用了 `k->set_class(k)`，最终将 `klassOop` 的 `_metadata` 指向了自己。但是在构建 `constantPoolKlass` 时，所传入的 `KlassHandle` 则是封装了 `klassKlass` 的 `handle`，因此最终所构建出来的 `constantPoolKlass` 所对应的 oop 的 `_metadata` 便指向前面所构建的 `klass`。最终 `constantPoolKlass` 内存布局如图 5.25 所示。

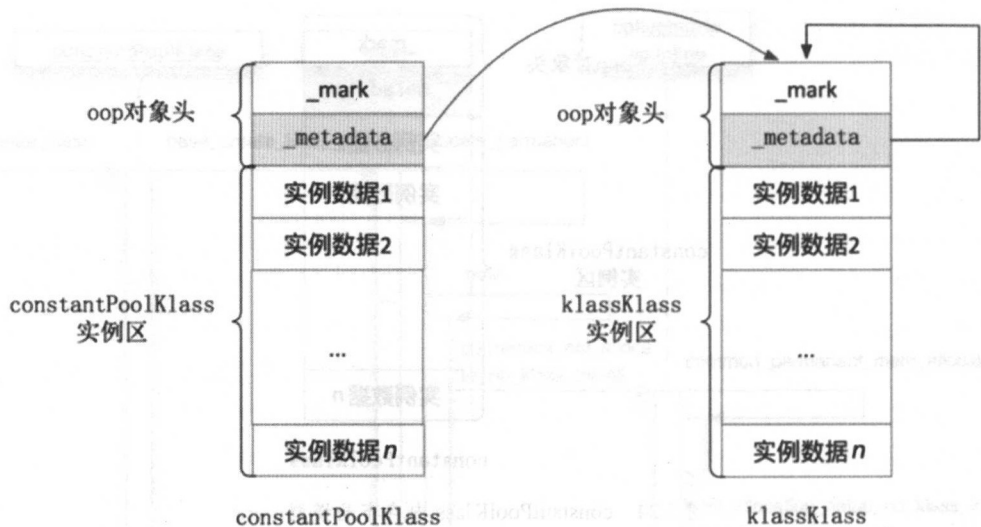


图 5.25 constantPoolKlass 内存布局

5.4.2 constantPoolOop 与 klass

刚刚分析了最终所构建出来的常量池 constantPoolKlass 的内存模型,其实 constantPoolKlass 正是 constantPoolOop 的类元信息,即 constantPoolOop 的 _metadata 指针应该指向 constantPoolKlass 对象实例。

而事实上,当 constantPoolOop 实例在构建过程中时,JVM 的确进行了这一步操作,这一步便是在图 5.26 中被椭圆框所框住的一步——post_allocation_install_obj_klass()。

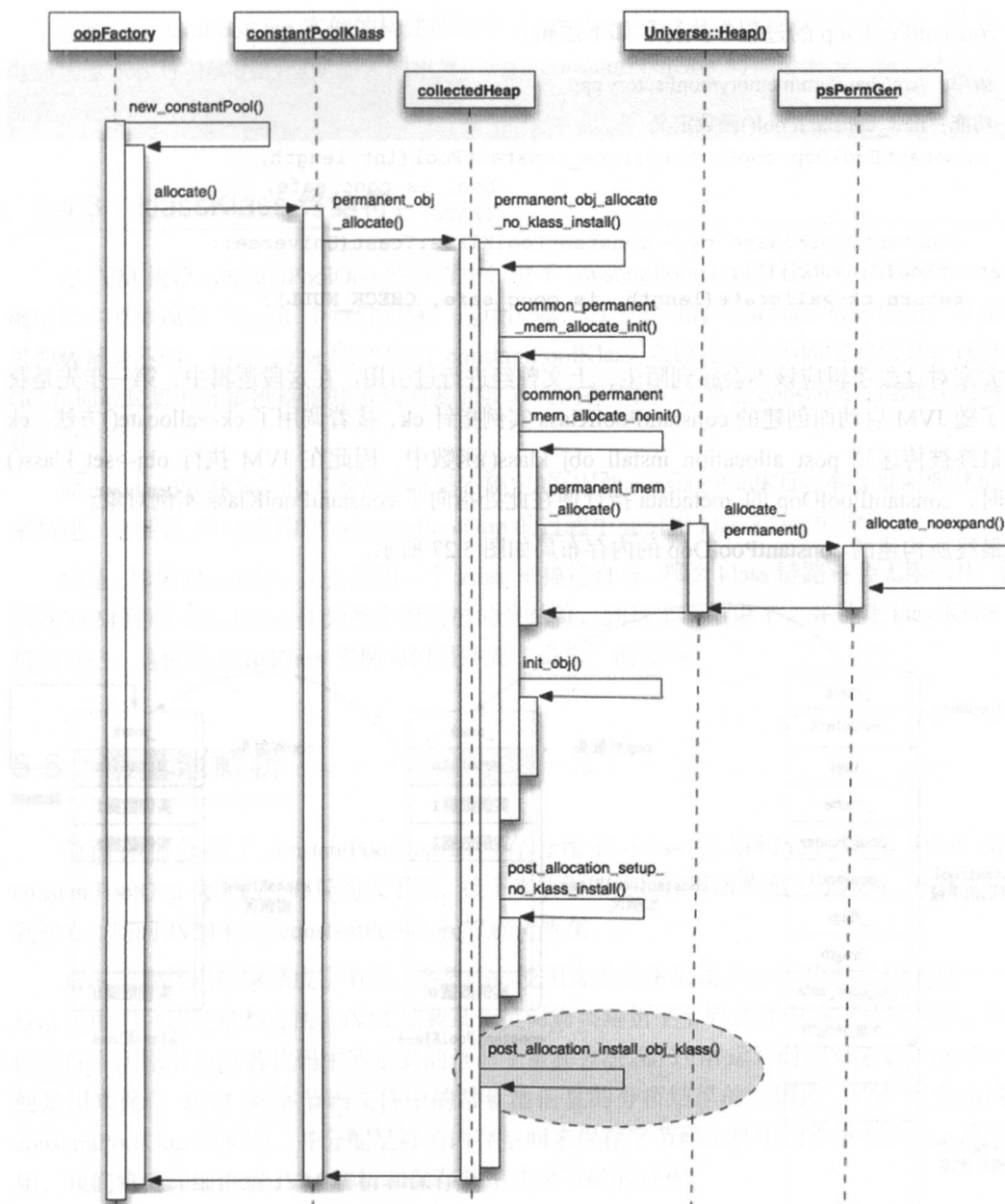


图 5.26 constantPoolOop 实例构建过程中设置_metadata

如果对前面构建 `klassKlass` 和构建 `constantOopKlass` 的过程很熟悉的话,应该清楚这一步实际上调用了 `obj->set_klass()` 这个函数,而该函数会将 `_metadata` 指针指向其对应的 `klass` 实例。JVM

创建 `constantPoolOop` 的过程中执行了如下逻辑：

清单：/src/share/vm/memory/oopFactory.cpp

功能：new_constantPool()函数定义

```
constantPoolOop oopFactory::new_constantPool(int length,
                                              bool is_conc_safe,
                                              TRAPS) {
    constantPoolKlass* ck = constantPoolKlass::cast(Universe::
constantPoolKlassObj());
    return ck->allocate(length, is_conc_safe, CHECK_NULL);
}
```

大家对这段逻辑应该不会感到陌生，上文曾经进行过引用。在这段逻辑中，第一步先是获取到了随 JVM 启动而创建的 `constantPoolKlass` 实例指针 `ck`，接着调用了 `ck->allocate()` 方法。`ck` 指针最终被传递到 `post_allocation_install_obj_class()` 函数中，因此在 JVM 执行 `obj->set_class()` 方法时，`constantPoolOop` 的 `_metadata` 指针便在此处指向了 `constantPoolKlass` 实例对象。

最终所构建的 `constantPoolOop` 的内存布局如图 5.27 所示。

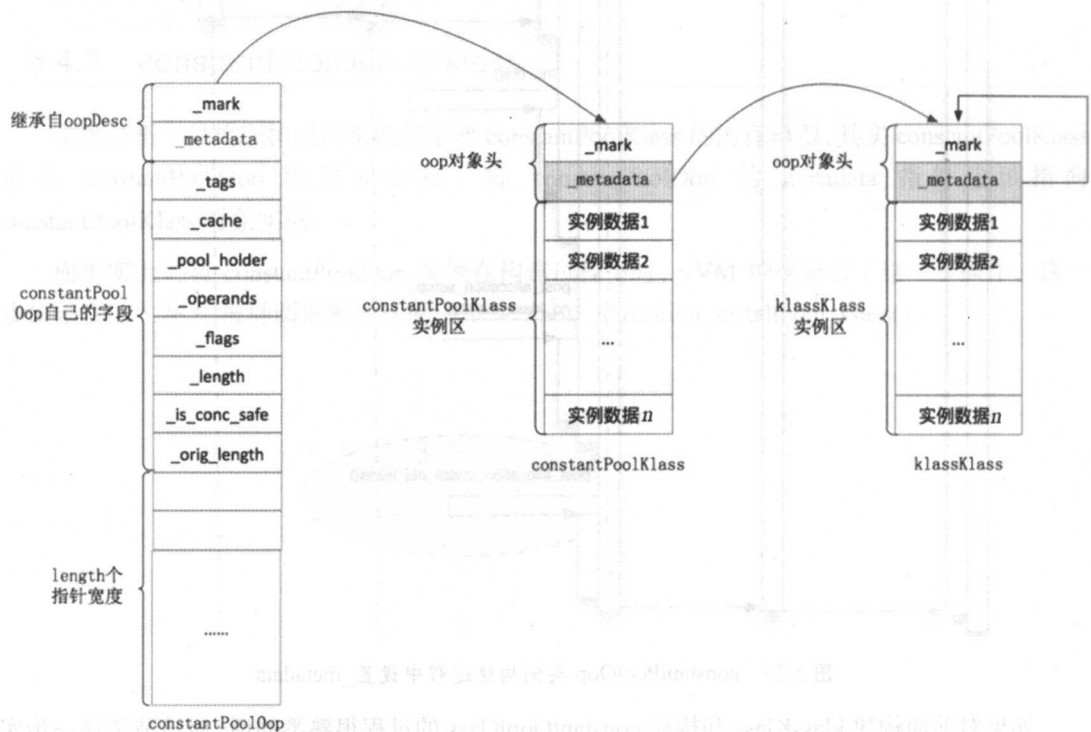


图 5.27 `constantPoolOop` 与 `constantPoolKlass`

至此, `constantPoolOop` 实例的构建便结束了。建议小伙伴们将这段逻辑熟记于心, 因为 JVM 内部其他 `oop` 对象实例的构建都大同小异, 明白 `constantPoolOop` 的构建原理, 便意味着明白了所有。

5.4.3 `klassKlass` 终结符

在 JVM 构建 `constantPoolOop` 的过程中, 由于 `constantPoolOop` 本身的大小不固定, 这种不确定性主要体现在“length 个指针宽度”这块区域, 因为不同的 Java class 被编译后, 常量池元素的数量会不同。因此 JVM 需要使用 `constantPoolKlass` 来描述这些不固定的信息, 这样最终 GC 在回收垃圾时才能准确地知道到底要回收多大内存空间。这便是 `constantPoolKlass` 的意义所在。

而 `constantPoolKlass` 的实例大小也是不确定的, 因此 `constantPoolKlass` 本身也需要其他 `klass` 来描述, 这便是 JVM 在构建 `constantPoolOop` 的过程中会引用 `klassKlass` 的原因。

但是, 如果 `klassKlass` 又去使用一个 `klass` 来描述自身, 那么 `klass` 链路将会无限引用下去, 因此 JVM 便将 `klassKlass` 作为整个引用链的终结符, 到这里就结束了, 并且让 `klassKlass` 自己指向自己。这便是 `klassKlass` 实例为何最终要“自指”的原因。

5.5 常量池解析

前面详细分析了 `constantPoolOop` 的内存分配和 `klass` 模型构建, 现在 JVM 完成了 `constantPoolOop` 全部内存布局的大手笔, 接下来要做的便是往里面填充实际数据。在这里不妨回到初心, 问问 JVM 构建 `constantPoolOop` 的意义所在。

Java 类源代码被编译成字节码, 字节码中使用常量池来描述 Java 类中的结构信息——包括 Java 类中的一切变量和方法。JVM 加载某个类时需要解析字节码文件中的常量池信息, 从字节码文件中还原出 Java 源代码中所定义的全部变量和方法。而 JVM 运行时的对象 `constantPoolOop` 便是用来保存 JVM 对字节码文件中的常量池信息的分析结果的。因此 JVM 需要先构建出 `constantPoolOop` 的实例, 并分配足够的内存空间来保存字节码文件中的常量池信息。从本节开始, 我们将会详细讲解 JVM 解析和保存常量池字节码的过程。

5.5.1 constantPoolOop 域初始化

在构建 constantPoolOop 的过程中，会执行 constantPoolClass::allocate() 函数，该函数主要干了 3 件事：

- ◎ 构建 constantPoolOop 对象实例。
- ◎ 初始化 constantPoolOop 实例域变量。
- ◎ 初始化 tag。

本节之前都在讲第一件事，这里讲第二件事。先看源码：

清单：/src/share/vm/oops/constantPoolClass.cpp

功能：constantPoolClass 域变量初始化

```
constantPoolOop constantPoolClass::allocate(int length, bool is_conc_safe,
TRAPS) {
    //...

    pool->set_length(length);
    pool->set_tags(NULL);
    pool->set_cache(NULL);
    pool->set_operands(NULL);
    pool->set_pool_holder(NULL);
    pool->set_flags(0);
    pool->set_orig_length(0);
    pool->set_is_conc_safe(is_conc_safe);
    //...
}
```

这部分逻辑执行完之后，constantPoolOop 的内存布局如图 5.28 所示。

注意：此时 constantPoolOop 域的 _tags 是 NULL，但是这个域变量对于 constantPoolOop 而言其实是非常重要的一个数据载体，_tags 实际上将会存放字节码常量池中的全部元素的标记。因此，下一步 JVM 便会对 _tags 进行一系列处理。

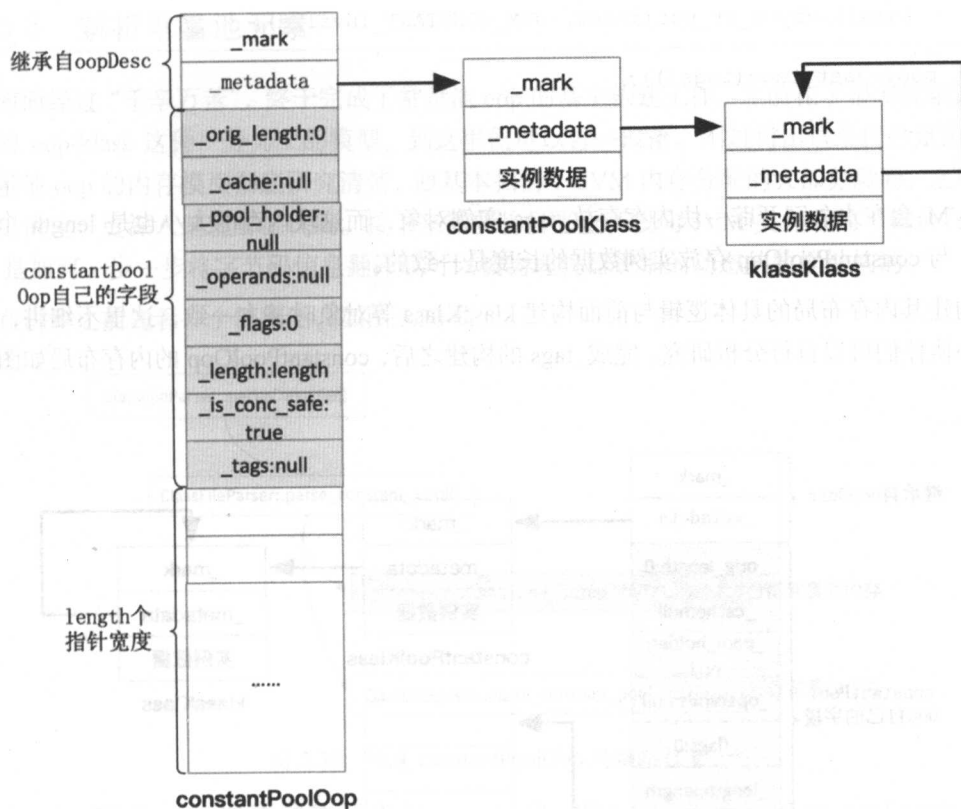


图 5.28 初始化 constantPoolOop 实例域变量

5.5.2 初始化 tag

在创建 constantPoolOop 的过程中，会为其_tags 域申请内存空间，这段逻辑还是在 constantPoolKlass::allocate()函数中实现，具体逻辑如下：

清单：/src/share/vm/oops/constantPoolKlass.cpp

功能：constantPoolKlass 域变量初始化

```
constantPoolOop constantPoolKlass::allocate(int length, bool is_conc_safe,
TRAPS) {
    //...

    typeArrayOop t_oop = oopFactory::new_permanent_byteArray(length, CHECK_NULL);
    typeArrayHandle tags (THREAD, t_oop);
    for (int index = 0; index < length; index++) {
```

```
tags()->byte_at_put(index, JVM_CONSTANT_Invalid);
}
pool->set_tags(tags());
//...
}
```

JVM 会在永久区开辟一块内存存放 `_tags` 实例对象，而这块内存的大小也是 `length` 个指针宽度，与 `constantPoolOop` 存放实例数据的长度是一致的。

构建其内存布局的具体逻辑与前面构建 `classKlass` 等对象时基本一致，这里不细讲，有兴趣的小伙伴们可以自行分析研究。完成 `_tags` 的构建之后，`constantPoolOop` 的内存布局如图 5.29 所示。

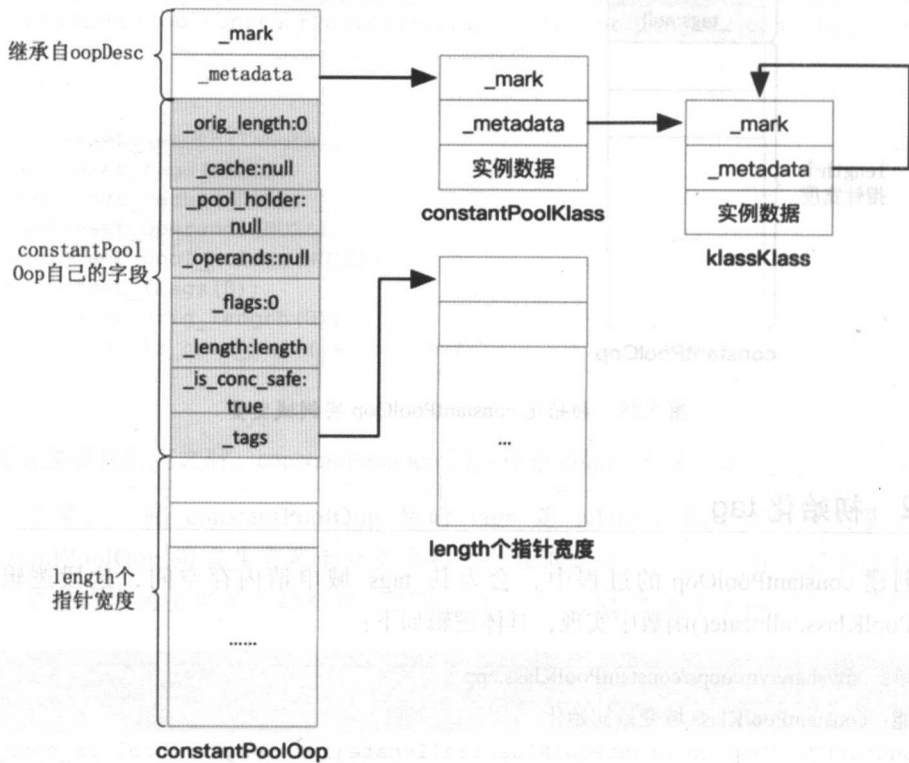


图 5.29 `constantPoolOop` 初始化 `_tags` 之后的内存布局

由于 `_tags` 所指向的实例对象刚刚构建，因此此时内存中没有实际数据。接下来 JVM 开始解析字节码文件的常量池元素，并逐个填充到这块内存区域。

5.5.3 解析常量池元素

前面经过“千辛万苦”，终于完成了常量池 oop 的基本构建工作。之前花了很多篇幅讲解常量池的 oop-klass 这种一分为二的模型，到这里便可以告一段落。不过付出这个代价是值得的，将常量池 oop 的内存模型彻底研究清楚，便基本扫清了 JVM 内存分配的大部分障碍，之后我们可以更快速、更深入地理解源代码。接下来需要将目光重新投向 JVM 类加载的“舞台”，研究 JVM 是如何一步一步将字节码信息翻译成可以被物理机器识别的动态的数据结构的。

在讲解之前先看一个链路图，如图 5.30 所示。

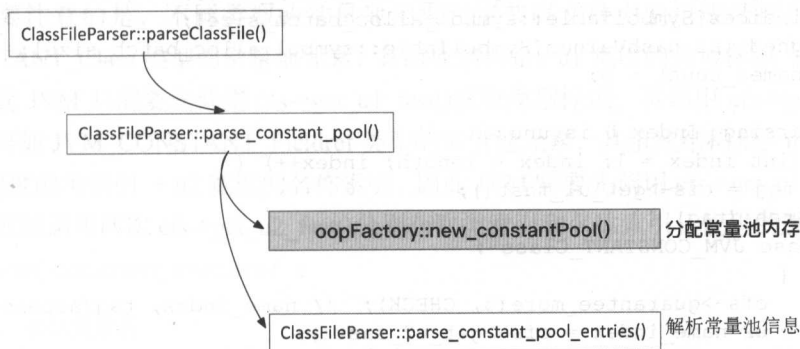


图 5.30 构建 constantPoolOop 的链路位置

在 classFileParser 中通过执行语句 `constantPoolOop constant_pool = oopFactory::new_constantPool()` 完成 constantPoolOop 的构建，前面都是在分析这部分逻辑。

接下来便开始解析常量池元素，为此 classFileParse 专门定义了一个函数：`parse_constant_pool_entries()`，其在链路图中的位置如图 5.31 所示。

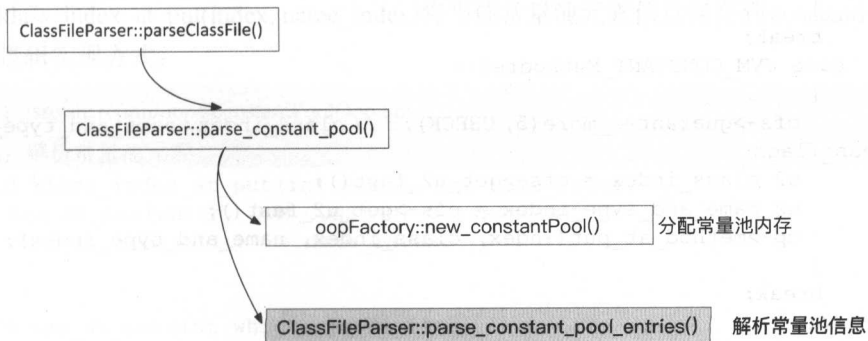


图 5.31 解析常量池元素的链路位置

先一睹 `parse_constant_pool_entries()` 函数的“芳容”：

清单：/src/share/vm/classfile/classFileParser.cpp

功能：常量池元素解析函数

```
void ClassFileParser::parse_constant_pool_entries(constantPoolHandle cp, int
length, TRAPS) {
    ClassFileStream* cfs0 = stream();
    ClassFileStream cfs1 = *cfs0;
    ClassFileStream* cfs = &cfs1;

    const char* names[SymbolTable::symbol_alloc_batch_size];
    int lengths[SymbolTable::symbol_alloc_batch_size];
    int indices[SymbolTable::symbol_alloc_batch_size];
    unsigned int hashValues[SymbolTable::symbol_alloc_batch_size];
    int names_count = 0;

    // parsing Index 0 is unused
    for (int index = 1; index < length; index++) {
        u1 tag = cfs->get_u1_fast();
        switch (tag) {
            case JVM_CONSTANT_Class :
            {
                cfs->guarantee_more(3, CHECK); // name_index, tag/access_flags
                u2 name_index = cfs->get_u2_fast();
                cp->klass_index_at_put(index, name_index);
            }
            break;
            case JVM_CONSTANT_Fieldref :
            {
                cfs->guarantee_more(5, CHECK); // class_index, name_and_type_index,
tag/access_flags
                u2 class_index = cfs->get_u2_fast();
                u2 name_and_type_index = cfs->get_u2_fast();
                cp->field_at_put(index, class_index, name_and_type_index);
            }
            break;
            case JVM_CONSTANT_Methodref :
            {
                cfs->guarantee_more(5, CHECK); // class_index, name_and_type_index,
tag/access_flags
                u2 class_index = cfs->get_u2_fast();
                u2 name_and_type_index = cfs->get_u2_fast();
                cp->method_at_put(index, class_index, name_and_type_index);
            }
            break;
            //...
        }
    }
}
```

本函数主要通过一个 for 循环处理所有的常量池元素，每次循环开始的时候，先执行语句 `u1 tag = cfs->get_u1_fast()`，从字节码文件中读取占 1 字节宽度的字节流。之所以这样，是因为在字节码文件中的常量池元素区，每一个常量池元素起始的 1 字节都用于描述常量池元素类型（这在前文分析过），JVM 解析常量池元素的第一步就是需要知道每个常量池元素的类型。

获取到常量池元素类型之后，通过 switch 条件表达式，对不同类型的常量池元素进行处理。这里以第一个 case `JVM_CONSTANT_Class` 语句为例进行说明。首先通过 `u2 name_index = cfs->get_u2_fast()` 获取类的名称索引，接着通过 `cp->klass_index_at_put(index, name_index)` 将当前常量池元素的类型和名称索引分别保存到 `constantPoolOop` 的 tag 数组和数据区。

这里需要注意的是，不同类型的常量池元素在字节码文件中的组成结构也不同，例如 `JVM_CONSTANT_Class` 类型的常量池元素，其组成结构是：u1 宽度的类型标识 + u2 宽度的名称索引，因此 JVM 只需要先调用 `cfs->get_u1_fast()` 获取类型标识，再调用 `cfs->get_u2_fast()` 获取其索引。再如 `JVM_CONSTANT_Fieldref` 类型的常量池元素，其组成结构是：u1 宽度的类型标识 + u2 宽度的类索引 + u2 宽度的名称索引，因此 JVM 需要先调用 `cfs->get_u1_fast()` 获取类型标识，再连续调用两次 `cfs->get_u2_fast()` 分别获取类索引和名称索引，代码的逻辑如下：

```
case JVM_CONSTANT_Fieldref :
{
    //获取类索引
    u2 class_index = cfs->get_u2_fast();

    //获取名称索引
    u2 name_and_type_index = cfs->get_u2_fast();

    //保存到 constantPoolOop 的 tag 和数据区中
    cp->field_at_put(index, class_index, name_and_type_index);
}
```

继续刚才类型为 `JVM_CONSTANT_Class` 的常量池元素的解析，获取到名称索引后，接着执行 `cp->klass_index_at_put(index, name_index)` 将当前常量池元素信息保存到 `constantPoolOop` 中。先看看其逻辑实现方式：

清单：/src/share/vm/oops/constantPoolOop.hpp

功能：解析常量池元素

```
void klass_index_at_put(int which, int name_index) {
    tag_at_put(which, JVM_CONSTANT_ClassIndex);
    *int_at_addr(which) = name_index;
}

void tag_at_put(int which, jbyte t) {
    tags()->byte_at_put(which, t);
}
```

在这个逻辑中，通过 `tag_at_put(which, JVM_CONSTANT_ClassIndex)` 将当前常量池元素的类型保存到 `constantPoolOop` 所指向的 `tag` 的对应位置的数组中，并通过 `*int_at_addr(which) = name_index` 将当前常量池元素的名称索引保存到 `constantPoolOop` 的数据区中对应的位置。

姑且将 `tag` 理解为数组（事实上也是，只不过被 `oop` 对象头包住了），当前常量池元素本身在字节码文件常量区中的位置索引将决定该常量池元素最终在 `tag` 数组中的位置，也决定该常量池元素的索引值最终在 `constantPoolOop` 的数据区的位置。

1. 类元素解析

为了说明概念，还是举个例子比较好，这里再次“祭出”前面所举的 `Iphone6s.java` 这个类，使用 `javap` 命令查看该类的结构如下：

清单：Iphone6s.java

作用：使用 Java 类描述 iPhone 6S 手机参数

```
Compiled from "Iphone6s.java"
public class Iphone6s extends java.lang.Object
    SourceFile: "Iphone6s.java"
    minor version: 0
    major version: 50
    Constant pool:
const #1 = class           #2;      // Iphone6s
const #2 = Asciz           Iphone6s;
const #3 = class           #4;      // java/lang/Object
const #4 = Asciz           java/lang/Object;
const #5 = Asciz           length;
const #6 = Asciz           I;
const #7 = Asciz           width;
const #8 = Asciz           height;
const #9 = Asciz           weight;
const #10 = Asciz          ram;
const #11 = Asciz          rom;
const #12 = Asciz          pixel;
const #13 = Asciz          <init>;
const #14 = Asciz          ()V;
const #15 = Asciz          Code;
const #16 = Method         #3.#17; // java/lang/Object."<init>":()V
const #17 = NameAndType    #13:#14; // "<init>":()V
const #18 = Field          #1.#19; // Iphone6s.length:I
const #19 = NameAndType    #5:#6; // length:I
const #20 = Field          #1.#21; // Iphone6s.width:I
const #21 = NameAndType    #7:#6; // width:I
const #22 = Field          #1.#23; // Iphone6s.height:I
const #23 = NameAndType    #8:#6; // height:I
```

```
const #24 = Field #1.#25; // Iphone6s.weight:I
const #25 = NameAndType #9:#6; // weight:I
const #26 = Field #1.#27; // Iphone6s.ram:I
const #27 = NameAndType #10:#6; // ram:I
const #28 = Field #1.#29; // Iphone6s.rom:I
const #29 = NameAndType #11:#6; // rom:I
const #30 = Field #1.#31; // Iphone6s.pixel:I
const #31 = NameAndType #12:#6; // pixel:I
const #32 = Asciz LineNumberTable;
const #33 = Asciz LocalVariableTable;
const #34 = Asciz this;
const #35 = Asciz LIphone6s;;
const #36 = Asciz SourceFile;
const #37 = Asciz Iphone6s.java;
```

由于第一号常量池元素的类型是 `JVM_CONSTANT_Class`, 还由于其在常量区中的位置是 1, 因此其最终在 `tag` 和 `constantPoolOop` 数据区中的位置也是 1。当 JVM 解析完这个常量池元素后, `constantPoolOop` 的内存布局如图 5.32 所示。

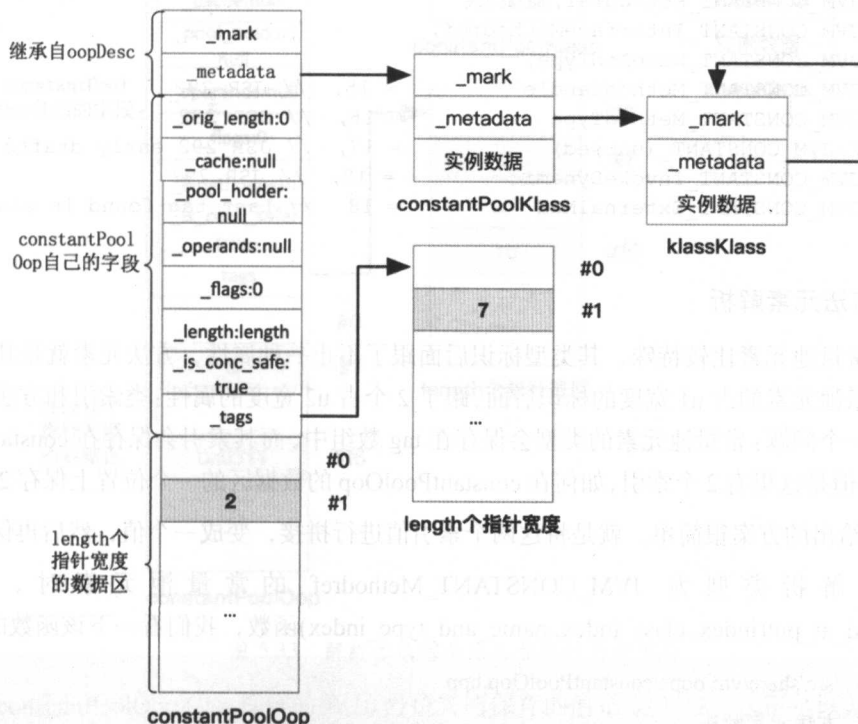


图 5.32 常量池解析示意

图 5.32 中的#0 和#1 表示数组下标位置, 其中在 constantPoolOop 的数据区的#1 号位置所存储的值为 2, 因为当前常量池元素(第 1 号常量池元素)的名称索引为 2。而 tag 的#1 号位置所存储的值为 7, 因为当前常量池元素的类型是 JVM_CONSTANT_Class, 该枚举值为 7。

这里贴出常量池元素类型枚举的定义:

清单: /src/share/vm/prims/jvm.h

功能: 常量池元素类型枚举

```
enum {
    JVM_CONSTANT_Utf8 = 1,
    JVM_CONSTANT_Unicode,           /* unused */
    JVM_CONSTANT_Integer,
    JVM_CONSTANT_Float,
    JVM_CONSTANT_Long,
    JVM_CONSTANT_Double,
    JVM_CONSTANT_Class,
    JVM_CONSTANT_String,
    JVM_CONSTANT_Fieldref,
    JVM_CONSTANT_Methodref,
    JVM_CONSTANT_InterfaceMethodref,
    JVM_CONSTANT_NameAndType,
    JVM_CONSTANT_MethodHandle       = 15, // JSR 292
    JVM_CONSTANT_MethodType         = 16, // JSR 292
    //JVM_CONSTANT_(unused)         = 17, // JSR 292 early drafts only
    JVM_CONSTANT_InvokeDynamic      = 18, // JSR 292
    JVM_CONSTANT_ExternalMax        = 18 // Last tag found in classfiles
};
```

2. 方法元素解析

有些常量池元素比较特殊, 其类型标识后面跟了不止一种属性, 方法元素就是其中一种。在方法常量池元素的占 u1 宽度的标识后面, 跟了 2 个占 u2 宽度的属性: 类索引和方法名索引。这便带来一个问题: 常量池元素的类型会保存在 tag 数组中, 而其索引会保存在 constantPoolOop 的数据区, 但是这里有 2 个索引, 如何在 constantPoolOop 的数据区的一个位置上保存 2 个值呢?

JVM 给出的方案很简单, 就是将这两个索引值进行拼接, 变成一个值, 然后再保存。

JVM 解析类型为 JVM_CONSTANT_Methodref 的常量池元素时, 会调用 cp->method_at_put(index, class_index, name_and_type_index) 函数, 我们看一下该函数的实现:

清单: /src/share/vm/oops/constantPoolOop.hpp

功能: 方法元素解析

```
void method_at_put(int which, int class_index, int name_and_type_index) {
    tag_at_put(which, JVM_CONSTANT_Methodref);
```

```

    *int_at_addr(which) = ((jint) name_and_type_index<<16) | class_index;
}

```

看到了没？这里将 `name_and_type_index` 左移了 2 字节，通过位或操作符与 `class_index` 完成两个值的拼接。在这里，位或操作符之所以能够实现两个数值的拼接，是因为这 2 个数值都只占 2 字节。

还是以 `Iphone6s.java` 为例，其常量池中第 16 号元素是 `Method`，其属性如下：

```
const #16 = Method      #3.#17; // java/lang/Object."<init>":()V
```

这表示其 `class_index` 是 3，而方法名索引是 17，当 JVM 将其解析完成后，`constantPoolOop` 的内存布局如图 5.33 所示。

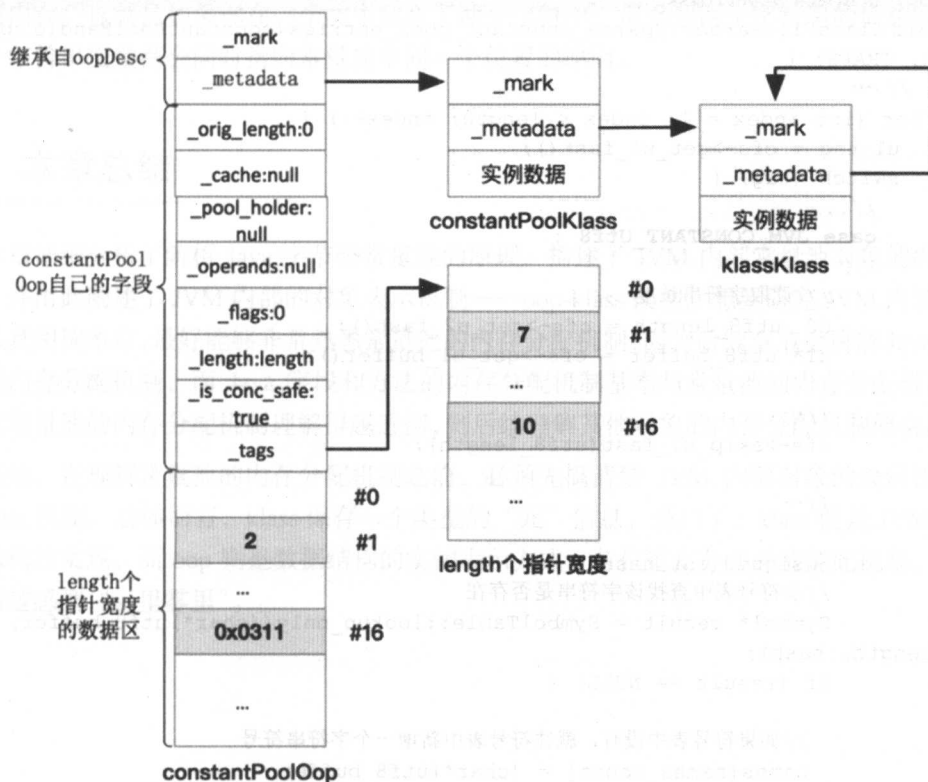


图 5.33 解析方法常量池元素后的内存布局

在 `constantPoolOop` 的数据区的第 16 号位置所保存的值是 `0x0311`，这正是该方法所对应的类索引值 3 和方法名索引 17 这两个数值拼接后的值。

3. 字符串元数据解析

对于常量池而言，字符串的概念比较广泛，并不单指字符串变量。类名、方法名、类型、this 指针名，等等，都可以看作是字符串，最终都会被 JVM 当作字符串处理，存储到符号区。

由于无论是 tag 还是 constantPoolOop 的数据区，一个存储位置只能存放一个指针宽度的数据，而字符串往往很大，因此 JVM 专门设计了一个“符号表”的内存区，tag 和 constantPoolOop 数据区内仅保存指针指向符号区。

JVM 对字符串的处理如下：

清单：/src/share/vm/classfile/classFileParser.cpp

功能：常量池元素解析函数

```
void ClassFileParser::parse_constant_pool_entries(constantPoolHandle cp, int
length, TRAPS) {
    // ...
    for (int index = 1; index < length; index++) {
        u1 tag = cfs->get_u1_fast();
        switch (tag) {
            //...
            case JVM_CONSTANT_Utf8 :
                {
                    // 读取字符串长度
                    u2 utf8_length = cfs->get_u2_fast();
                    u1* utf8_buffer = cfs->get_u1_buffer();

                    //...
                    cfs->skip_u1_fast(utf8_length);

                    //...

                    unsigned int hash;
                    //从符号表中查找该字符串是否存在
                    Symbol* result = SymbolTable::lookup_only((char*)utf8_buffer,
utf8_length, hash);
                    if (result == NULL) {

                        //如果符号表中没有，就往符号表中新增一个字符串符号
                        names[names_count] = (char*)utf8_buffer;
                        lengths[names_count] = utf8_length;
                        indices[names_count] = index;
                        hashValues[names_count++] = hash;
                        if (names_count == SymbolTable::symbol_alloc_batch_size) {
                            SymbolTable::new_symbols(cp, names_count, names, lengths,
indices, hashValues, CHECK);
                        }
                    }
                }
            }
        }
    }
}
```

```

        names_count = 0;
    }
    } else {
        cp->symbol_at_put(index, result);
    }
}
break;
//...
}

```

以上代码给出了一个基本思路，即字节码文件中的字符串常量池元素最终都会被保存到符号表中。为了节省内存，JVM 会先判断符号表中是否存在相同的字符串，如果已经存在，则不会申请内存。这就是如果你在一个类中定义了两个字符串，但是这两个字符串的值相同，最终这两个字符串变量都会同时指向常量池中同一个位置的原因。

5.6 本章总结

本章详细分析了解析 Java 字节码常量池的原理，描述了 JVM 内部常量池对象的内存分配机制，并由此阐述了 JVM 内部的对象表示机制——oop-klass 模型。想要研究 JVM 内核的道友应当认真阅读本章，最好能够非常熟悉常量池的内存分配机制，因为后续章节会讲解 Java 字段、方法的内存分配机制，而 Java 字段和方法的内存分配机制基本与常量池的内存分配机制类似，因此对常量池的内存分配机制理解得越透彻，则后续理解其他对象的内存分配机制便会越轻松。

当然，在理解常量池的内存分配机制之前，必须先搞清楚 JVM 内部对象的表示机制——oop-klass 机制。总体而言，klass 保存一个类型的“元”信息，说白了，klass 便是 JVM 内部对数据结构的实现，而 oop 则是数据结构的实例表示方式。各位道友务必弄清楚此机制，否则越到后面越感到“云里雾里”。

第 6 章

类变量解析

本章摘要

- ◎ Java 类变量解析的原理
- ◎ 计算机基础——偏移量与内存对齐
- ◎ Java 类与字段的对齐与补白
- ◎ Java 字段的继承机制
- ◎ 使用 HSDB 查看运行时的 Java 类结构

JVM 从 Java 类的字节码文件中解析出常量池信息后，便将 Java 类的变量和方法基本信息读取进内存，保存在 `constantPoolOop` 的 tag 和数据区。但是 tag 与数据区中的数据仍然是非结构化的数据，根据这些数据并不能直观地描述 Java 类结构，因此 JVM 需要进一步解析。这便是本章要讲的主要内容。按照本书的“惯例”，为了加强前后思路的连贯性，防止思路被分支所干扰，下面给出类解析的总体地图，如图 6.1 所示。

本章开始讲解 JVM 解析 Java 类中所定义的变量的技术实现细节。

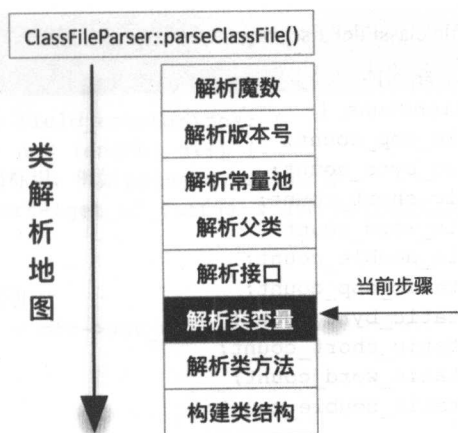


图 6.1 类解析的步骤

6.1 类变量解析

在 `ClassFileParser::parseClassFile()` 函数中，解析完常量池、父类和接口后，接着便调用 `parse_fields()` 函数解析类变量信息：

清单：/src/share/vm/classfile/classFileParser.cpp

功能：解析类变量

```
void ClassFileParser::parseClassFile(constantPoolHandle cp, int length,
TRAPS) {
    // ...
    struct FieldAllocationCount fac = {0,0,0,0,0,0,0,0,0,0};
    objArrayHandle fields_annotations;
    typeArrayHandle fields = parse_fields(cp, access_flags.is_interface(),
    &fac, &fields_annotations,
        CHECK_(nullHandle));
    //...
}
```

在调用 `parse_fields()` 函数之前，先定义了一个变量 `fac`，这里的 `FieldAllocationCount` 是一个结构体类型，声明如下：

清单：/src/share/vm/classfile/classFileParser.cpp

功能：FieldAllocationCount 结构体

```
struct FieldAllocationCount {
    unsigned int static_oop_count;
    unsigned int static_byte_count;
    unsigned int static_short_count;
    unsigned int static_word_count;
    unsigned int static_double_count;
    unsigned int nonstatic_oop_count;
    unsigned int nonstatic_byte_count;
    unsigned int nonstatic_short_count;
    unsigned int nonstatic_word_count;
    unsigned int nonstatic_double_count;
};
```

通过声明可知，FieldAllocationCount 结构体类型的变量实例将会记录 5 种静态（static）类型变量的数量和 5 种对应的非静态类型的变量的数量，这 5 种变量类型分别是：

- ◎ Oop，引用类型
- ◎ Byte，字节类型
- ◎ Short，短整型
- ◎ Word，双字类型
- ◎ Double，浮点型

在 parse_fields() 函数里面，会分别统计 static 和非 static 的这 5 种变量的数量，后面 JVM 为 Java 类分配内存空间时，会根据这些变量的数量计算所占内存大小。可能有小伙伴会问：Java 语言所支持的实际数据类型远不止这 5 种呀，例如 boolean、float、int，等等，为什么 JVM 只统计这 5 种呢？这是因为在 JVM 内部，除了引用类型，所有内置的基本类型都使用剩余的 4 种类型来表示，因此想知道一个 Java 类的域变量需要占用多少内存，只需要分别统计这几种类型的数量即可。

下面一起看看 JVM 对 Java 类域变量的具体解析逻辑：

清单：/src/share/vm/classfile/classFileParser.cpp

功能：ClassFileParser::parse_fields()

```
typeArrayHandle ClassFileParser::parse_fields(constantPoolHandle cp, bool
is_interface,
```

```
struct FieldAllocationCount *fac,
objArrayHandle* fields_annotations,
```

```
TRAPS) {
    //...
    // 获取 Java 类域变量的数量
```



```

u2 length = cfs->get_u2_fast();

int index = 0;
typeArrayHandle field_annotations;
for (int n = 0; n < length; n++) {
    //读取变量的访问标识, 例如 private|public 等
    jint flags = cfs->get_u2_fast() & JVM_RECOGNIZED_FIELD_MODIFIERS;
    //...

    //读取变量名称索引
    u2 name_index = cfs->get_u2_fast();
    //...

    //读取类型索引
    u2 signature_index = cfs->get_u2_fast();
    //...

    //读取变量属性
    u2 attributes_count = cfs->get_u2_fast();
    //...

    fields->short_at_put(index++, access_flags.as_short());
    fields->short_at_put(index++, name_index);
    fields->short_at_put(index++, signature_index);
    fields->short_at_put(index++, constantvalue_index);

    //判断变量类型
    BasicType type = cp->basic_type_for_signature_at(signature_index);
    FieldAllocationType atype;
    if ( is_static ) {
        switch ( type ) {
            case T_BOOLEAN:
            case T_BYTE:
                fac->static_byte_count++;
                atype = STATIC_BYTE;
                break;
            case T_LONG:
                //...

        default:
            assert(0, "bad field type");
        }
    }

    fields->short_at_put(index++, atype);
    fields->short_at_put(index++, 0);

```

```
fields->short_at_put(index++, generic_signature_index);
}
```

上面对 `ClassFileParser::parse_fields()` 的主要逻辑进行了注释，通过注释可以知道，JVM 解析 Java 类域变量的逻辑是：

- ◎ 获取 Java 类中的变量数量。
- ◎ 读取变量的访问标识。
- ◎ 读取变量名称索引。
- ◎ 读取变量类型索引。
- ◎ 读取变量属性。
- ◎ 判断变量类型。
- ◎ 统计各类型数量。

在 Java 类所对应的字节码文件中，有专门的一块区域保存 Java 类的变量信息（在前文详细讲解过），字节码文件会依次描述各个变量的访问标识、名称索引、类型索引和属性。由于每个变量的访问标识、名称索引、类型索引和属性数量都占用 2 字节（u2），因此在这段逻辑中，只需依次调用 `cfs->get_u2_fast()` 即可。

解析完一个变量的属性后，调用 `fields->short_at_put()` 函数将属性保存到 `fields` 所代表的内存区域中。`fields` 是在 `ClassFileParser::parse_fields()` 方法一开始就申请的内存，如下：

```
u2 length = cfs->get_u2_fast();
typeArrayOop new_fields =
    oopFactory::new_permanent_shortArray(
        length*instanceClass::next_offset, CHECK_(nullHandle));
```

如果你非常认真地研究了前文所讲的常量池对象 `constantPoolOop` 的内存申请与分配机制，那么这里对于 `typeArrayOop` 的内存分配机制自然一看就明白，因此这里不关注其实现的具体过程。但是，有一个问题却非常值得关注：这里到底申请了多大的内存？

`oopFactory::new_permanent_shortArray()` 函数的第一个入参决定了最终所分配的内存大小，而该入参并没有直接给出，而是一个表达式：

```
length * instanceClass::next_offset
```

`length` 自然是指 Java 类中的变量数量，而 `instanceClass::next_offset` 在 `instanceClass.hpp` 文件中定义，是一个枚举：

清单：/src/share/vm/oops/instanceClass.hpp

功能：FieldOffset 枚举定义

```
enum FieldOffset {
```

```

access_flags_offset = 0,
name_index_offset  = 1,
signature_index_offset = 2,
initval_index_offset = 3,
low_offset         = 4,
high_offset        = 5,
generic_signature_offset = 6,
next_offset        = 7
};

```

由该定义可知，`instanceKlass::next_offset` 的值为 7。

由此也可知，最终所申请的内存大小是 $7 * \text{length}$ ，由于 `length` 类型是 `u2`，占 2 字节，因此所申请的内存大小实际上相当于 14 个变量所占用的总字节。假设 Java 类中一共有 5 个变量，则 JVM 需要为其申请 $14 * 5 = 70$ 字节的内存空间，来存放变量的属性。

这里仍然有一点需要注意，字节码文件对 Java 类变量的描述维度与 JVM 内存中的映像并不相同，字节码文件仅仅描述了变量的访问标识、名称索引、类型索引、属性，而 JVM 内存映像除此之外，还描述了每一个变量的偏移量和泛型索引。而关于偏移量的话题，正是下一节的主题。

在遍历解析各个变量属性的循环后面，JVM 会取出每一个变量的类型进行判断，并根据类型来统计上文所提到的 5 大类型（静态和非静态各 5 种类型）的总数量。

6.2 偏移量

解析完字节码文件中 Java 类的全部域变量信息后，JVM 计算出 5 种数据类型各自的数量，并据此计算各个变量的偏移量。

6.2.1 静态变量偏移量

先看静态类型变量，其逻辑如下：

清单：/src/share/vm/classfile/classFileParser.cpp

功能：静态变量类型定义

```

typeArrayHandle ClassFileParser::parseClassFile(constantPoolHandle cp, bool
is_interface,
    struct FieldAllocationCount *fac,
    objArrayHandle* fields_annotations,
    TRAPS) {

```

```
//...
//定义各种偏移量
int static_field_size = 0;
int next_static_oop_offset;
int next_static_double_offset;
int next_static_word_offset;
int next_static_short_offset;
int next_static_byte_offset;
int next_static_type_offset;

//...

//获取起始偏移量
next_static_oop_offset = instanceMirrorKlass::offset_of_static_fields();

//计算 5 大类型的静态变量的偏移量
next_static_double_offset = next_static_oop_offset +
                             (fac.static_oop_count * heapOopSize);
/...
next_static_word_offset   = next_static_double_offset +
                             (fac.static_double_count * BytesPerLong);
next_static_short_offset  = next_static_word_offset +
                             (fac.static_word_count * BytesPerInt);
next_static_byte_offset   = next_static_short_offset +
                             (fac.static_short_count * BytesPerShort);
next_static_type_offset   = align_size_up((next_static_byte_offset +
                             fac.static_byte_count ), wordSize );

//计算全部静态变量所占的字节数
static_field_size = (next_static_type_offset -
                     next_static_oop_offset) / wordSize;

//...
}
```

这段逻辑很简单，先拿到起始偏移量，接着分别根据各个静态类型变量的偏移量计算总偏移量。计算顺序是：

- ◎ static_oop
- ◎ static_double
- ◎ static_word
- ◎ static_short
- ◎ static_byte

每一种“下游”数据类型的偏移量都依赖其“上游”数据类型所占的字节宽度及数量。

JVM 为何要记录每一个变量的偏移量呢？其实，这与静态变量的存储机制和访问机制有关。对于 JDK 1.6 而言，静态变量存储在 Java 类在 JVM 中所对应的镜像类 Mirror 中，当 Java 代码访问静态变量时，最终 JVM 也是通过设置偏移量进行访问。

6.2.2 非静态变量偏移量

相比于静态变量偏移量，非静态变量的偏移量计算稍显复杂。逻辑如下：

清单：/src/share/vm/classfile/classFileParser.cpp

功能：ClassFileParser::parse_fields()

```
typeArrayHandle ClassFileParser::parseClassFile(constantPoolHandle cp, bool
is_interface,
```

```
struct FieldAllocationCount *fac,
objArrayHandle* fields_annotations,
```

```
TRAPS) {
```

```
    //...
```

```
    //获取父类非静态字段大小（以字节为单位）
```

```
    int nonstatic_field_size = super_klass() == NULL ? 0 :
```

```
super_klass->nonstatic_field_size();
```

```
    //...
```

```
    //...
```

```
    //①.计算非静态字段起始偏移量
```

```
    first_nonstatic_field_offset = instanceOopDesc::base_offset_in_bytes() +
                                nonstatic_field_size * heapOopSize;
```

```
    next_nonstatic_field_offset = first_nonstatic_field_offset;
```

```
    //...
```

//②.计算 **nonstatic_double_offset** 和 **nonstatic_oop_offset** 这 2 种非静态类型变量的起始偏移量

```
    //分配策略不同，这 2 种变量的起始偏移量也不同
```

```
    if( allocation_style == 0 ) {
```

```
        // Fields order: oops, longs/doubles, ints, shorts/chars, bytes
```

```
        next_nonstatic_oop_offset = next_nonstatic_field_offset;
```

```
        //...
```

```
    }
```

```
    //...
```

```
    //处理压缩类型
```

```
    if( nonstatic_double_count > 0 ) {
```

```
        int offset = next_nonstatic_double_offset;
```

```
        next_nonstatic_double_offset = align_size_up(offset, BytesPerLong);
```

```
        if( compact_fields && offset != next_nonstatic_double_offset ) {
```

```

    int length = next_nonstatic_double_offset - offset;
    //...
}

//...
//③.计算剩余 3 种类型变量的起始偏移量: nonstatic_word_offset,
nonstatic_short_offset, nonstatic_byte_offset
next_nonstatic_word_offset = next_nonstatic_double_offset +
    (nonstatic_double_count * BytesPerLong);
next_nonstatic_short_offset = next_nonstatic_word_offset +
    (nonstatic_word_count * BytesPerInt);
next_nonstatic_byte_offset = next_nonstatic_short_offset +
    (nonstatic_short_count * BytesPerShort);

//④.计算对齐补白空间
int notaligned_offset;
if( allocation_style == 0 ) {
    notaligned_offset = next_nonstatic_byte_offset + nonstatic_byte_count;
} else { // allocation_style == 1
    next_nonstatic_oop_offset = next_nonstatic_byte_offset +
nonstatic_byte_count;
    if( nonstatic_oop_count > 0 ) {
        next_nonstatic_oop_offset = align_size_up(next_nonstatic_oop_offset,
heapOopSize);
    }
    notaligned_offset = next_nonstatic_oop_offset + (nonstatic_oop_count *
heapOopSize);
}
next_nonstatic_type_offset = align_size_up(notaligned_offset,
heapOopSize );

//⑤.计算补白后非静态变量所需要的内存空间总大小
nonstatic_field_size = nonstatic_field_size + ((next_nonstatic_type_offset
    - first_nonstatic_field_offset)/heapOopSize);

//...
//5 大类型变量的偏移量都计算完, 现在开始遍历所有字段, 分别计算各类型具体变量的偏移值, 这
里包括静态的和非静态的
int len = fields->length();
for (int i = 0; i < len; i += instanceKlass::next_offset) {
    int real_offset;
    FieldAllocationType atype = (FieldAllocationType) fields->ushort_at(i +
        instanceKlass::low_offset);
    switch (atype) {
        case STATIC_OOP:
            real_offset = next_static_oop_offset;
            next_static_oop_offset += heapOopSize;

```

```

    //...
}

//保存偏移量
fields->short_at_put(i + instanceKlass::low_offset,
extract_low_short_from_int(real_offset));
fields->short_at_put(i + instanceKlass::high_offset,
extract_high_short_from_int(real_offset));
}

//... //计算 oop_map_count
const unsigned int total_oop_map_count =
compute_oop_map_count(super_klass, nonstatic_oop_map_count,
first_nonstatic_oop_offset);

//...
}

```

如上面代码所注释的那样，非静态类型变量的偏移量计算逻辑可以分为 5 个步骤：

- (1) 计算非静态变量起始偏移量。
- (2) 计算 `nonstatic_double_offset` 和 `nonstatic_oop_offset` 这 2 种非静态类型变量的起始偏移量。
- (3) 计算剩余 3 种类型变量的起始偏移量：`nonstatic_word_offset`、`nonstatic_short_offset` 和 `nonstatic_byte_offset`。
- (4) 计算对齐补白空间。
- (5) 计算补白后非静态变量所需要的内存空间总大小。

JVM 的内存管理模型在所有编程语言中属于比较复杂的一种，而对于一个给定的 Java 类，其主要的内存空间便是其非静态类型的变量所占用的空间（另一部分是虚拟方法分发表），因此理解了这部分内存分配策略，JVM 的内存管理模型便理解了一半，故这部分内容十分重要。下面分析非静态类型变量偏移量的计算逻辑。

1. 计算非静态变量起始偏移量

要计算非静态变量的偏移量，首先需要知道从哪里开始偏移。

本节多次讲到了偏移量，那么什么是偏移量呢？对于非静态类型的变量，其偏移量是相对于未来即将 new 出来的 Java 对象实例在 JVM 内部所对应的 `instanceOop` 对象实例首地址的偏移位置。

前面描述常量池对象时提到过，在 JVM 内部，使用 `oop-klass` 这种一分为二的模型去描述

一个对象，常量池对象本身便是这种模型。对于 Java 类，JVM 内部同样使用这种一分为二的模型去描述，对应的 oop 类是 instanceOopDesc，对应的 klass 类是 instanceKlass。在 oop-klass 模型中，oop 模型主要存储对象实例的实际数据，而 klass 模型则主要存储对象的结构信息和虚函数方法表，说白了就是描述类的结构和行为。

当 JVM 加载一个 Java 类时，会首先构建对应的 instanceKlass 对象，而当 new 一个 Java 对象实例时，则会构建出对应的 instanceOop 对象。instanceOop 对象主要由 Java 类的成员变量组成，而这些成员变量在 instanceOop 中的排列顺序，便由各种变量类型的偏移量决定。

在 HotSpot 内部，任何 oop 对象都包含对象头，因此实际上非静态变量的偏移量要从对象头的末尾开始计算。Java 类实例在堆内存中的起始偏移量如图 6.2 所示。

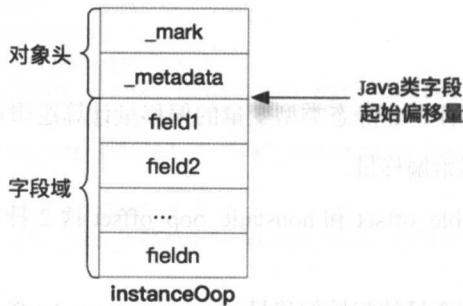


图 6.2 Java 类字段的起始偏移量

知道了偏移量的含义，现在来看看 JVM 的实现。源代码里有如下逻辑：

清单：/src/share/vm/classfile/classFileParser.cpp

功能：计算非静态变量起始偏移量

```
typeArrayHandle ClassFileParser::parseClassFile(constantPoolHandle cp, bool
is_interface,
                                     struct FieldAllocationCount *fac,
                                     objArrayHandle* fields_annotations,
TRAPS) {
    //...
    //获取父类非静态字段大小（以字节为单位）
    int nonstatic_field_size = super_klass() == NULL ? 0 :
super_klass->nonstatic_field_size();
    //...
    //计算非静态字段起始偏移量
    first_nonstatic_field_offset = instanceOopDesc::base_offset_in_bytes() +
                                     nonstatic_field_size * heapOopSize;
    //...
```

}

这里首先调用 `instanceOopDesc::base_offset_in_bytes()` 方法，该方法实现如下：

清单：/src/share/vm/oops/instanceOop.hpp

功能：计算对象头大小

```
static int base_offset_in_bytes() {
    return UseCompressedOops ?
        klass_gap_offset_in_bytes() :
        sizeof(instanceOopDesc);
}
```

如果没有启用压缩策略，则最终返回 `instanceOopDesc` 类型大小。`instanceOopDesc` 继承自 `oopDesc`，而 `oopDesc` 类型大小在前文已经计算出来了，是两个指针宽度。假设 JVM 运行在 64 位架构上，则这个值是 16。而如果启用了压缩策略，则在 64 位架构上，该值为 12，这是因为无论是否开启压缩策略，`oop._mark` 作为一个指针是不会被压缩的，任何时候都会占用 8 字节，而 `oop._klass` 则会受压缩策略是否开启的影响，若开启压缩策略，则 `_klass` 仅会占用 4 字节，所以在 64 位架构上开启压缩策略的情况下，`oop` 对象头总共占用 12 字节的内存空间。

Java 类是面向对象的，继承是面向对象编程的 3 大特性之一，除了 `java.lang.Object` 类以外，所有的 Java 类都显式或隐式地继承了某个父类，而字段继承和方法继承则构成了继承的全部内涵。

如果说继承是目标，那么字段在子类中的内存占用则是技术手段。子类必须将父类中所定义的非静态字段信息全部复制一遍，才能实现字段继承的目标。因此，在计算子类非静态字段的起始偏移量时，必须将父类可被继承的字段的内存大小考虑在内。具体而言，子类的非静态字段起始偏移量，在计算完 `oop` 对象头的大小后，还需要为父类的可被继承的字段预留空间。

Hotspot 将父类可被继承的字段的内存空间安排在子类所对应的 `oop` 对象头的后面，因此最终一个 Java 类中非静态字段的起始偏移位置在父类被继承的字段域的末尾，如图 6.3 所示。

2. 内存对齐与字段重排

在上面第一步计算出 Java 类字段的起始偏移量后，接下来就能基于这个起始偏移量计算出 Java 类中所有字段的偏移量。不过在这之前，需要先研究这样一个问题——内存对齐。

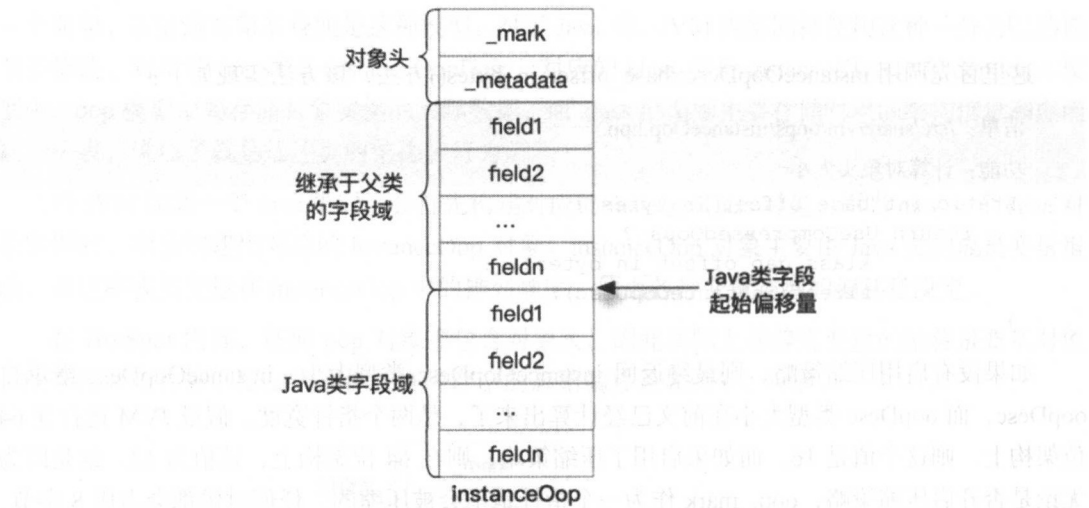


图 6.3 Java 类字段的起始偏移量

因为 Java 类字段的偏移地址与内存对齐有脱不开的关系，JVM 为了处理内存对齐，颇费了一番心思，甚至不惜将字段进行重排。

1) 什么是内存对齐

内存对齐与数据在内存中的位置有关。如果一个变量的内存起始地址正好等于其长度的整数倍，则这种内存分配就被称作自然对齐。

在 32 位 x86 平台上，基本的对齐规则如表 6.1 所示。

表 6.1 32 位 x86 平台上的内存对齐规则

变 量 类 型	长 度	对 齐 规 则
char	1 字节	按 1 字节对齐
short	2 字节	按 2 字节对齐
int	4 字节	按 4 字节对齐

举个例子，在 32 位 CPU 下，假设一个 int 整型变量的内存地址为 0x00000008，那么该变量就是自然对齐的。

2) 为什么要字节对齐

现代计算机中内存空间都是按照字节划分的，也即内存的粒度细分到存储单元，而一个存储单元恰恰包含 8 个比特，正好是 1 字节(byte)，因此从理论上讲似乎对任何类型的变量的访问可以从任何地址开始。

但实际情况恰恰相反，各个硬件平台在对存储空间的处理上有很大的不同。一些平台对某些特定类型的数据只能从某些特定地址开始存取。例如有些架构的 CPU 在访问一个没有进行对齐的变量时会发生错误，那么在这种架构下进行编程就必须保证字节对齐。

例如，在某些硬件架构上，如果一个 int 整型变量的内存地址为 0x00000003，那么当在程序中对该变量进行读写时，硬件就会报错。要验证 CPU 硬件平台是否支持非对齐状态的变量读写，只需通过如下一个 C 程序进行测试：

清单：示例程序

作用：验证 CPU 硬件平台对内存对齐的支持

```
char ch[2];

char *p = &ch[0];
int i = *(int *)p;

p = &ch[1];
i = *(int *)p;
```

这段程序很简单，先声明一个大小为 2 的 char 类型数组，接着将该数组的第 1 个元素（从 1 开始计数）的内存地址传递给指针 p，再通过变量 i 从数组的第 1 个元素所在的内存位置开始连续读取 4 字节（即一个 int 类型的宽度）。接着再从数组的第 2 个元素所在的内存位置开始连续读取 4 字节数据。

如果 ch 数组的首地址恰好是 4 字节的整数倍，则第一次读取 4 字节能够成功，但是第二次一定会失败，因为第二次读取时，起始地址一定不是 4 字节的整数倍。

某些平台会支持非对齐内存位置的数据读写，硬件不会抛出异常，但是最常见的是，如果不按照其平台要求对数据存放进行对齐，会在存取效率上有所损失。比如有些平台每次读都是从偶地址开始，如果一个 int 型（假设为 32 位系统）存放在偶地址开始的地方，那么一个读周期就可以读出这 32 比特，而如果存放在奇地址开始的地方，就需要 2 个读周期，并对两次读出的结果的高低字节进行拼凑才能得到该 32 比特数据。显然在读取效率上下降很多。

例如，假设一个整型变量的内存地址不是 4 字节对齐，其内存位置位于 0x00000002，则 CPU 需要进行两次内存访问才能读取到该变量的值：

- ◎ 第一次从 0x00000002 和 0x00000003 这两个连续的内存存储单元中读取一个 short 类型宽度的数据。
- ◎ 第二次则从 0x00000004 和 0x00000005 这两个连续的内存单元中再读取一个 short 类型宽度的数据。

将这 2 次读取的数据进行组合之后才能得到所要的数据。

如果该变量的内存位置是 0x00000003，则 CPU 可能需要 2 次甚至 3 次内存访问才能将该变量的值读取出来。CPU 访问两次的方案是：

- ◎ 第一次从 0x00000001~0x00000004 这 4 个连续的内存单元中读取一个 int 类型宽度的数据。
- ◎ 第二次则从 0x00000005~0x00000008 这 4 个连续的内存单元中读取一个 int 类型宽度的数据。

将两次内存访问的结果剔除首尾多出来的字节，拼凑出目标结果。

而 CPU 访问三次的方案是：

- ◎ 第一次从 0x00000003 这个内存单元上读取一个 char 类型宽度的数据。
- ◎ 第二次从 0x00000004 和 0x00000005 这两个连续的内存单元中读取一个 short 类型宽度的数据。
- ◎ 第三次则从 0x00000006 这个内存单元上读取一个 char 类型宽度的数据。

三次都读取完成后，再将这 3 次读取的结果进行组合得到整型数据。

如果这样的程序工作在多线程环境中，则还需要进行总线级别的原子操作，以防止出现脏数据，从而造成程序的严重错误。

而如果这个整型变量的内存地址是自然对齐的，则 CPU 只需要一次内存访问就能读取数据，并且还是原子性的，效率 and 安全性无疑是最高。

所以，由于以上 2 种情况，一种是程序健壮性，一种是高性能，均需要各种类型数据按照一定的规则在空间上排列，而不是顺序地一个一个地排放，从而对数据进行对齐。

3) 什么时候需要考虑对齐

无论是 C/C++ 这样的编译性语言，还是 Java/C# 这样的高级语言，一般情况下都不需要开发者去考虑对齐的问题，因为编译器或者虚拟机会自动将数据进行对齐补白。不过如果你是在设计底层协议或者开发硬件驱动程序，这时候就必须要考虑对齐的事情了。

下面这个简单的例子能够验证 C/C++ 的编译器 gcc 对内存对齐的支持：

清单：示例程序

作用：gcc 编译器自动处理数据对齐示例

```
#include <stdio.h>
```

```
void test();
```

```
int main(){
```

```

    test(6);
}

void test(int x){
    char c = 65;
    short s = 16;
    int i = 8;
    printf("%d\n", x + i);
}

```

本例很简单，在 `test()` 函数里声明和定义了 3 种不同类型的变量，之所以要声明这几个不同类型的变量，就是为了测试 gcc 编译器的对齐处理能力。

使用 gcc 编译器将这段 C 程序编译成汇编程序，如下所示：

清单：示例程序

作用：gcc 编译器自动处理数据对齐示例

```

main:
    pushq    %rbp
    movl$6, %edi
    callq    _test
test:
    pushq    %rbp
    subq$32, %rsp

    movl%edi, -4(%rbp) // 获取入参
    movb$65, -5(%rbp) // 为变量 c 赋值
    movw$16, -8(%rbp) // 为变量 s 赋值
    movl$8, -12(%rbp) // 为变量 i 赋值

    movl-4(%rbp), %edi
    addl-12(%rbp), %edi
    movl%edi, -16(%rbp)      ## 4-byte Spill
    movq%rax, %rdi
    movl-16(%rbp), %esi      ## 4-byte Reload
    movb$0, %al
    callq    _printf
    movl%eax, -20(%rbp)      ## 4-byte Spill
    addq$32, %rsp
    popq%rbp
    retq

```

在这段汇编程序的 `test` 段中，中间有几条分别为 `test()` 函数中的局部变量进行赋值的指令，其中变量 `c` 的堆栈位置是 `-5(%rbp)`，因为 `c` 是字节类型，因此只占 1 个内存存储单元。变量 `c` 之后的局部变量是 `short` 类型的 `s`，由于 `short` 类型占 2 个内存存储单元，因此按常理 `s` 的堆栈位

置应该是 $-7(\%rbp)$ ，但是编译器却将变量分配在了 $-8(\%rbp)$ 这个位置，其实在这里，编译器便是对变量进行了对齐处理。由于 `short` 类型的宽度是 2 字节，因此其内存首地址必须是 2 字节的整数倍，如果将变量 `s` 的偏移地址放在 $-7(\%rbp)$ 这个位置，很显然不符合自然对齐的原则。

下面这个结构体的例子则很经典，能够使你在不了解汇编语言的情况上，直观地观察到编译器对内存对齐处理的支持：

清单：示例程序

作用：C 语言的 `struct` 内存对齐

```
#include <stdio.h>

struct A
{
    int a;
    char b;
    short c;
};

struct B
{
    char b;
    int a;
    short c;
};

int main(){
    int a = sizeof(struct A);
    int b = sizeof(struct B);
    printf("%d\n", a);
    printf("%d\n", b);
}
```

在这段程序中，声明了 2 个结构体类型 `A` 和 `B`，注意，这两个结构体中所包含的变量数量和变量类型完全一致，唯一不一致的是变量的声明顺序。

结构体所占的内存大小，往往是其中所有变量所占内存的总和，按照这个逻辑，则本例中的两个结构体的大小应该都是一样大小。在 32 位机器上，以上几种数据类型的长度如下：

- ◎ `Char`，1（有符号无符号都相同）
- ◎ `Short`，2（有符号无符号都相同）
- ◎ `Int`，4（有符号无符号都相同）

`a` 是 `int` 类型，宽度是 4 字节；`b` 是 `char` 类型，宽度是 1 字节；`c` 是 `short` 类型，宽度是 2 字节。所以这两个结构体所占的内存大小应该是 7 字节。然而结果并不是这样。运行 `main()` 函数，

打印的结果如下:

`sizeof(struct A)`的值为 8。

`sizeof(struct B)`的值却是 12。

很奇怪,两个结构体的大小竟然没有一个是 7。这是因为 gcc 编译器对变量进行了内存对齐。

对于结构体 A,其变量类型的顺序是 `int->char->short`,在内存分配时,先为 `int` 类型的变量 a 分配 4 字节内存。接着为 `char` 类型的 b 变量分配内存。由于 `char` 仅占 1 字节内存,因此其实并无内存对齐要求,所以变量 b 的内存可以直接跟在变量 a 之后。假设变量 a 的内存起始地址标记为 $4x$ (即 4 字节的整数倍),则 b 的内存地址为 $4(x+1)$ (为了方便描述问题,暂时忘记堆栈空间的生长方向到底是往高低址还是往低地址吧)。

接着为 `short` 类型的 c 变量分配内存。如果不考虑内存对齐,则变量 c 应当位于变量 b 的后面,则变量 c 的内存地址应该为 $4(x+1)+1$ 。但是由于 `short` 类型的数据的对齐原则是按 2 字节对齐,而 $4(x+1)+1$ 这样的内存地址很显然并不能被 2 整除,不符合按 2 字节对齐的要求,所以编译器就自作主张,将其又往后移动了一个字节,将变量 c 分配在 $4(x+1)+2$ 的内存地址。这样一来,给人的感觉是 `char` 类型的变量 b 似乎占据了 2 个内存单元。其实变量 b 并没有占据 2 个存储单元,仍然只需要一个存储单元的内存空间,而其后面的另一个多出来的存储单元是用于对齐的补白空间。因此,最终 `struct A` 需要占用的内存空间总大小就是 8 字节。其内存空间布局如图 6.4 所示。

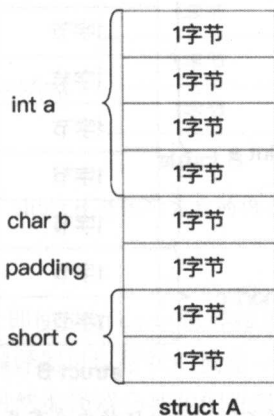


图 6.4 struct A 的内存布局

再来分析 struct B。其变量类型的顺序为 char->int->short。分配内存时，先为 char 类型的 b 变量分配内存空间。b 变量只需要一个存储单元便足够了。

接着为 int 类型的变量 a 分配内存。假设变量 b 的内存地址为 $4x$ （即 4 字节的整数倍），这里解释一下，变量 b 的内存地址之所以也是 4 字节的整数倍，这是由于编译器不仅保证各种变量、结构、复合结构的数据类型是内存对齐的，还会保证堆、堆栈的内存地址也是自然对齐的。所以无论结构体 B 被分配在哪里，其内存起始地址一定是 4 的整数倍。假设变量 a 的内存位置紧跟在变量 b 的后面，则 a 的内存地址应该是 $4x+1$ （为了方便描述问题，暂时忘记堆栈空间的增长方向到底是往高地址还是往低地址吧），因为变量 b 只需要 1 个内存单元即可。但是由于内存对齐的需要，变量 a 需要按 4 字节对齐，所以肯定不能分配在 $4x+1$ 这个内存位置，那怎么办呢？很简单，往后移动 3 个字节，补齐至 4 字节即可。移动 3 字节后的内存位置是 $4(x+1)$ ，这下能被 4 整除了，所以就分配在这里了。

接着为 short 类型的变量 c 分配内存。由于变量 a 占 4 字节内存空间，因此变量 c 的起始内存地址自然就是相对于变量 a 的起始地址 $4(x+1)$ 再往后移动 4 个字节，所以变量 c 的起始内存地址变成了 $4(x+2)$ 。

现在，内存布局是如图 6.5 所示的样子。

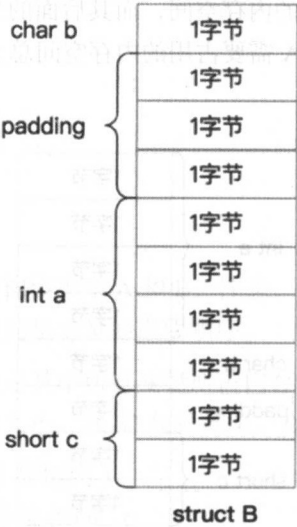


图 6.5 struct B 的内存布局

在这种内存布局下，struct B 总共占有 10 字节内存空间。但是在此刻，内存对齐的规则又发挥作用，具体发挥的对象就是 struct B 类型本身。

在编译原理中，不仅是基本的数据类型要求做到自然对齐，连结构体这样的复合结构类型也需要整体进行自然对齐。其实，数据类型的对齐并不是为了方便自己，而是为了方便别人。例如，struct B 中的 char 类型的变量 b 后面补白了 3 字节，是为了让紧跟其后的 int 类型的 a 变量能够自然对齐。同理，struct B 也需要考虑让其后面的变量能够自然对齐。由于 struct B 也不知道其后面会跟什么类型的变量，因此便按照默认的 4 字节进行对齐。这么做使得后续在处理数据对齐时的逻辑变得简单。

由于编译器要将 struct 结构体按照 4 字节对齐，因此最终 struct B 的内存空间会被扩展到 12 字节，最终的内存布局如图 6.6 所示。

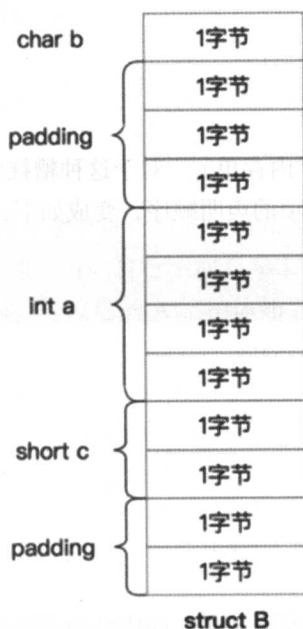


图 6.6 struct B 整体对齐后的内存布局

4) 人工优化结构体内存空间

通过上面的两个例子可以看到，即使是如 c/c++ 这样接近底层的编程语言，平时在开发中也不需要关注内存对齐的问题，因为编译器可以帮我们搞定一切。但是上面那 2 个结构体的例子却向我们透漏出一个强烈的信号：虽然编译器会帮我们处理好内存对齐的问题，但是编译器并不是万能的，并不会使用除了内存对齐以外的其他优化技巧。上面例子中的 A 和 B 两个结构体所包含的成员项是完全相同的，所不同的仅仅是成员项的声明顺序不同而已，结果就造成两者在内存空间利用率上的巨大差异。

假设结构体的成员项数量增多，并且声明完全是无序的，那么内存的利用率将更加糟糕，例如下面这个结构体：

清单：示例程序

作用：C 语言的 struct 内存对齐

```
struct A
{
    char b;
    int a;

    char b2;
    int a2;

    char b3;
    int a3;
};
```

像这种结构体，将会占用 24 个内存单元。对于这种糟糕的内存使用率，程序员完全可以主动优化，例如，更改下结构体成员项的声明顺序，变成如下：

清单：示例程序

作用：C 语言的 struct 内存对齐

```
struct A
{
    char b;
    char b2;
    char b3

    int a;
    int a2;
    int a3;
};
```

修改后的结构体只需要占用 16 个内存单元，比优化之前的结构体一下子少占用 8 字节的内存空间，这种内存空间的节省量相当可观。

请记住这个示例，下面讲解 JVM 的内存分配策略时，与此会有很大关系。

所以，在进行 C/C++ 程序开发时，一个优秀的程序员还是应当主动关注内存对齐的问题。在这方面，一种可以遵循的设计技巧是：在定义结构体类型中的成员项时，按照类型大小从小到大依次声明，如此便能够尽量减少中间的填补空间。

除了这种设计技巧，还有一种主动的以空间换时间的策略，显式地声明填补空间，例如对上面例子中的 A 结构体进行如下处理：

```
struct A
{
    int a;
    char b;
    char reserved;//声明一个补白字节
    short c;
};
```

在变量 **b** 后面多声明一个成员项，其类型也是 **char**。这个成员对程序没有明显的意义，仅起到填补空间以达到字节对齐的目的。当然，即使不加这个成员项，编译器也会自动填补对齐，我们自己加上它只是起到显式的提醒作用。

假设 **A** 结构体的结构是这样的：

```
struct A
{
    char a;
    int b;
};
```

由于变量 **b** 是 **int** 类型，需要按 4 字节进行对齐，因此编译器会自动在变量 **a** 的后面填补 3 字节以对齐。如果你不够信赖编译器，可以自行添加 3 字节的补白空间，添加方式就是声明一个元素数量为 3 的 **char** 类型的数组。修改后的 **A** 结构体如下：

```
struct A
{
    char a;
    char reserved[3];
    int b;
};
```

5) Java 类字段对齐的要求

Java 类的字段最终也是要保存到堆内存中的，也需要被 CPU 频繁地读写，并且 Java 类的变量类型最终也会映射成 CPU 硬件平台架构所能支持的数据类型，因此 Java 类字段在内存中的位置也必须满足自然对齐的要求。尤其是 Java 语言作为一门跨平台的编程语言，可以运行在众多的硬件平台上，更应该兼容各种异构平台对数据对齐的要求，否则万一运行在某个硬件平台上，而该平台压根儿不支持非对齐内存的访问，那么 Java 虚拟机很可能就会可怜地因为这个低级的原因而直接崩溃。

因此，JVM 在设计上就必须使其内部 5 大类型的数据都满足内存对齐的原则。Java 类中包含八大基本数据类型，每种数据类型所占用的内存空间大小总体上与 C/C++ 中的基本数据类型的宽度相等，如表 6.2 所示。

表 6.2 Java 类基本数据类型所占用的内存大小

Java 基本数据类型	占用空间(bit)	占用空间(byte)
boolean	8	1
byte	8	1
char	16	2
short	16	2
int	32	4
float	32	4
long	64	8
double	64	8

除了这 8 种基本数据类型，还有另外一种类型，就是引用。引用数据类型所占用的内存大小与不同的硬件平台以及是否开启压缩策略有关。在 32 位平台上，引用数据类型占用 4 字节内存空间，而在 64 位平台上，如果开启了压缩策略，则占用 4 字节内存空间，否则便占用 8 字节内存空间。

Java 语言中的 8 种基本数据类型与引用类型对应 JVM 内部所定义的 5 大数据类型。它们的对应关系如表 6.3 所示。

表 6.3 Java 数据类型与 JVM 内部 5 种数据类型的对应关系

Java 基本数据类型	JVM 内部数据类型	数据宽度
引用类型	oop	4 字节/8 字节
boolean/byte	byte	1 字节
char/short	short	2 字节
int/float	word	4 字节
long/double	double	8 字节

JVM 必须确保其内部这 5 种基本数据类型都能够自然对齐，即确保其内存地址能够被其所占用的字节宽度所整除。解决数据类型自然对齐的手段无非是内存补白，JVM 自然也少不了这一手（事实上除了这法子也没别的法子了），但是 JVM 却技高一筹，不仅使用了补白，还祭出了另一件法宝——字段重排。

JVM 的字段重排策略主要包括下面 2 点：

- ◎ 将相同类型的字段组合在一起。
- ◎ 按照 double->word->short->byte-oop 的顺序依次分配。

第一点，将相同类型的字段组合在一起，究其原因，是因为这样更能节省内存空间。还记得上面用 C 语言写的那个结构体的例子吗？使用 C 语言编写一个结构体时，结构体中成员项声明的顺序不同，则结构体的大小也随之不同。这种规律对 Java 字段同样适用。看下面这个简单的 Java 类：

清单：示例程序
作用：Java 语言的内存对齐

```
class A{
    byte b;
    long l;
    byte b2;
    int i;
}
```

如果没有字段重排，则 JVM 为了让各种类型的字段做到自然对齐，最终只能按照如图 6.7 所示这种补白的方式来分配内存（省略 oop 对象头）。

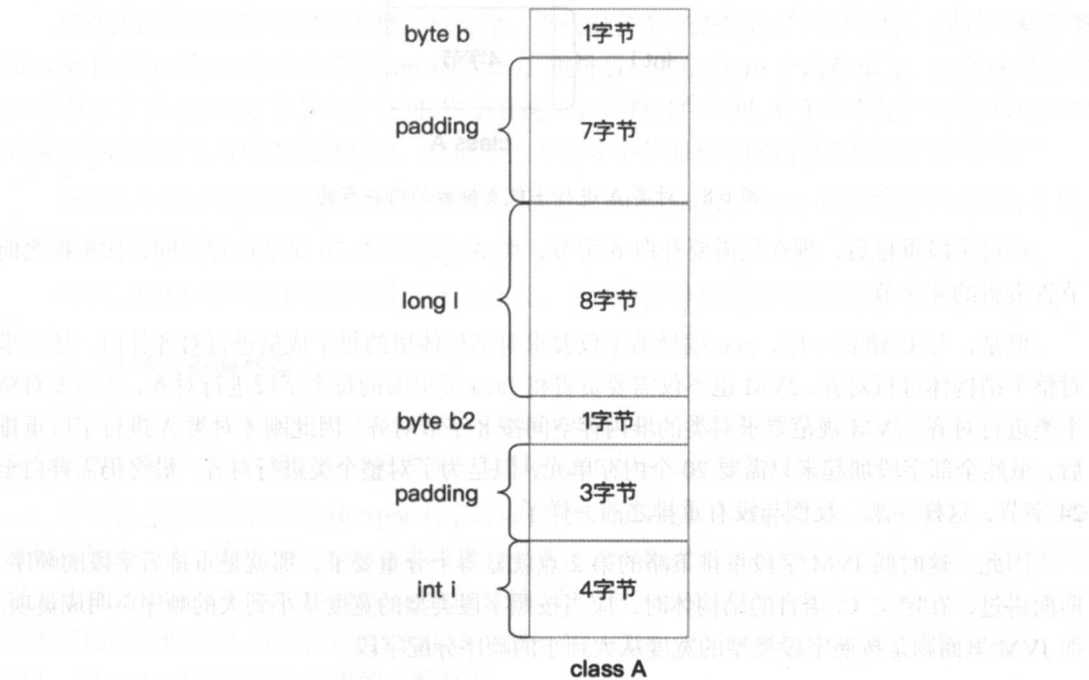


图 6.7 未进行字段重排时类 A 的内存布局

这样的内存布局需要占用 24 字节的内存空间。

如果将相同类型的字段组合在一起进行内存分配，并且假设按照 `byte -> long -> int` 的顺序分配内存，则最终所分配的内存空间布局如图 6.8 所示。

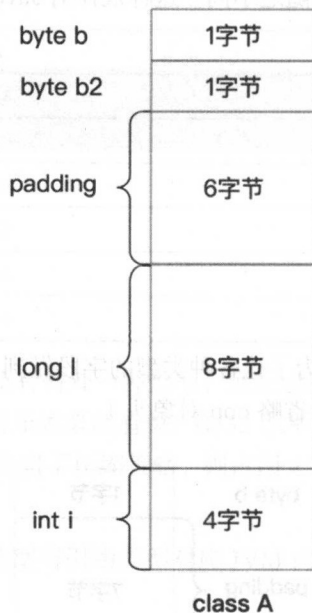


图 6.8 对类 A 进行字段重排后的内存布局

经过字段重排后，现在只需要补白 6 字节，类 A 总共只占 20 字节内存空间，比重排之前节省宝贵的 4 字节。

但是，与 C 语言一样，gcc 编译器不仅要求对结构体里的每个成员进行对齐补白，还要求对整个结构体进行对齐，JVM 也不仅需要负责将 Java 类里面的每个字段进行对齐，还需要对整个类进行对齐。JVM 规范要求对类的堆内存空间按 8 字节对齐，因此刚才对类 A 进行字段重排后，虽然全部字段加起来只需要 20 个内存单元，但是为了对整个类进行对齐，最终仍需补白至 24 字节，这样一来，反倒与没有重排之前一样了。

因此，这时候 JVM 字段重排策略的第 2 点就显得十分重要了，那就是重排后字段的顺序。前面讲过，在定义 C 语言的结构体时，应当按照字段类型的宽度从小到大的顺序声明成员项。而 JVM 里面则是按照字段类型的宽度从大到小的顺序分配字段。

现在按照 `long -> int -> byte` 的顺序对类 A 的字段空间重新排列，重排后的内存空间布局如图 6.9 所示。

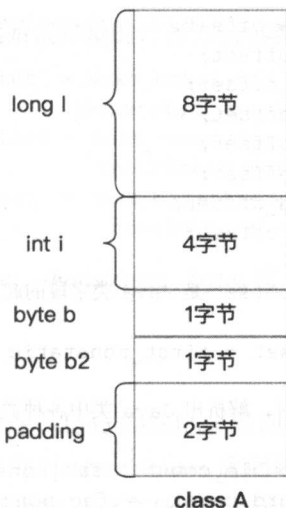


图 6.9 class A 按指定顺序重排后的内存布局

现在按照新的顺序对数据类型进行排序, 不仅对各个字段进行了内存对齐, 同时对整个类按 8 字节进行对齐 (暂时没考虑 oop 对象头), 重排后一共仅占 16 个内存单元, 比优化之前的 24 字节节省了宝贵的 8 字节内存空间。别小看这 8 字节的内存空间, 对于一个生产环境中的 JVM, 其需要加载成千上万个类实例对象, 这成千上万个 8 字节加起来的内存空间便十分可观。

通过这个例子也可以知道, JVM 要按照 long -> int -> short -> byte 的顺序对字段进行排列, 实在是有其道理的。

当然, JVM 对字段重排的优化还不止如此, 下文会在分析源码时再次讲到优化策略。

3. 计算变量偏移量

本书虽然研究理论, 但更关注 JVM 内部的实现细节, 这是本书的宗旨。上面关于 JVM 内存分配的原理讲了那么多, 但终究要窥一窥 HotSpot 内部究竟是如何实现的。

其实通过前面所阐述的 Hotspot 对字段内存分配的策略, 不难推导出一种计算 Java 类各字段偏移量的方法。既然 Hotspot 将字段进行了重排, 将相同类型的字段存储在一起, 那么便可以先计算出其内部 5 大类型字段的起始偏移量。每一种类型都包含零或多个 Java 类字段, 基于该类型的起始偏移量, 便可逐个计算出该类型所对应的每一个具体的 Java 类字段的偏移量。事实上, Hotspot 也就是这么实现的。看源码:

```
清单: /src/share/vm/classfile/classFileParser.cpp
```

```
功能: ClassFileParser::parseClassFile()
```

```
int next_nonstatic_oop_offset;
```

```

int next_nonstatic_double_offset;
int next_nonstatic_word_offset;
int next_nonstatic_short_offset;
int next_nonstatic_byte_offset;
int next_nonstatic_type_offset;
int first_nonstatic_oop_offset;
int first_nonstatic_field_offset;
int next_nonstatic_field_offset;

//first_nonstatic_field_offset 是 Java 类字段的起始偏移量, 这里将 next_nonstatic_
field_offset 也指向起始偏移量
next_nonstatic_field_offset = first_nonstatic_field_offset;

//前置流程解析 Java 类常量池时, 解析出 Java 类中各种类型的字段数量, 这里分别获取这 5 种类
型的字段的数量
unsigned int nonstatic_double_count = fac.nonstatic_double_count;
unsigned int nonstatic_word_count  = fac.nonstatic_word_count;
unsigned int nonstatic_short_count  = fac.nonstatic_short_count;
unsigned int nonstatic_byte_count   = fac.nonstatic_byte_count;
unsigned int nonstatic_oop_count    = fac.nonstatic_oop_count;

//根据不同的顺序策略, 计算 oop 或者 long 的起始偏移量
if( allocation_style == 0 ) {
    // Fields order: oops, longs/doubles, ints, shorts/chars, bytes
    next_nonstatic_oop_offset  = next_nonstatic_field_offset;
    next_nonstatic_double_offset = next_nonstatic_oop_offset +
                                   (nonstatic_oop_count * heapOopSize);
} else if( allocation_style == 1 ) {
    //如果 allocation_style 是 1, 则字段分配顺序是 longs/doubles、ints、shorts/chars、
    bytes、oops
    //由于先分配 long 类型字段所以 long 类型的字段的起始偏移量自然就是整个 Java 类字段的起始
    偏移量
    next_nonstatic_double_offset = next_nonstatic_field_offset;
}

```

HotSpot 提供了好几种重排顺序选项。如果 `allocation_style` 的值是 0, 则按照 `oops -> longs/doubles -> ints -> shorts/chars -> bytes` 的顺序为字段分配内存空间; 如果 `allocation_style` 的值是 1, 则最先分配 `longs/doubles`, 最后分配 `oops`。这里仅讨论后一种情况。

根据上面的源码, 此时其实 `longs/doubles` 类型的起始偏移量已经计算出来了, 这个偏移量就是整个 Java 类的起始偏移量。

分配完 `longs/doubles` 类型之后接着分配 `ints` 类型, 说白了就是计算 `ints` 的起始偏移量。`ints` 的起始偏移量一定位于 `longs` 内存空间的末尾, 所以 `ints` 的起始偏移量的计算方法是:

`longs/doubles` 的起始偏移量 + `longs` 的宽度 * `longs` 的数量。

ints 之后的各种数据类型的起始偏移量的计算方法也类似, 我们看 HotSpot 的实现:

```
next_nonstatic_word_offset = next_nonstatic_double_offset +
                             (nonstatic_double_count * BytesPerLong);
next_nonstatic_short_offset = next_nonstatic_word_offset +
                              (nonstatic_word_count * BytesPerInt);
next_nonstatic_byte_offset = next_nonstatic_short_offset +
                             (nonstatic_short_count * BytesPerShort);
```

通过这段代码, HotSpot 将 ints、shorts/chars、bytes 这 3 种字段类型的起始偏移量也计算出来。

这里要注意两点:

- ◎ 无论字段重排是哪种顺序, longs/doubles 后面跟的一定是 ints, 而 ints 后面所跟的一定是 shorts/chars, 并且 shorts/chars 后面跟的一定是 bytes。
- ◎ 由于 longs/doubles 字段进行了对齐处理, 所以其末尾的下一个内存地址一定是 8 字节的整数倍。对于这样的内存地址, 其也一定是 4 字节的整数倍, 所以将 ints 紧跟在 longs/doubles 字段后面, ints 的字段也就天然是对齐的。同理, ints 后面的 shorts/chars 以及 shorts/chars 后面的 bytes 字段也一定是天然对齐的。

此时的内存布局如图 6.10 所示。

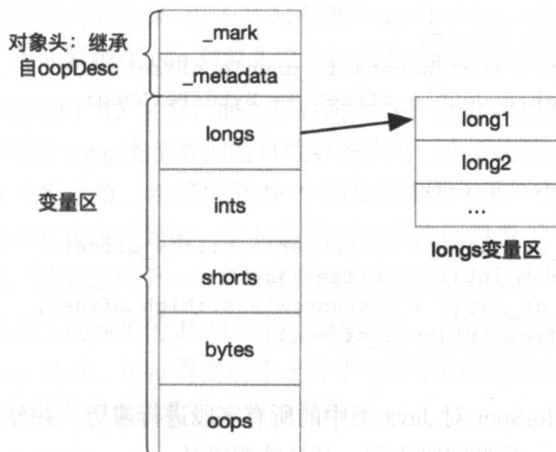


图 6.10 内部 5 大类型的起始偏移量

完成了 JVM 内部 5 大类型数据的起始偏移量计算之后, 接着就可以计算每种类型所对应的 Java 类中的字段的具体偏移量了。计算方法很简单, 将 Java 类中的字段按照其所属的 5 大类型的起始偏移量进行顺序排列即可。看 HotSpot 的源码实现:

清单：/src/share/vm/classfile/classFileParser.cpp

功能：ClassFileParser::parseClassFile()

```
// 获取 Java 类中非静态字段的数量
int len = fields->length();
for (int i = 0; i < len; i += instanceKlass::next_offset) {
    int real_offset;
    FieldAllocationType atype = (FieldAllocationType) fields->ushort_at(i +
instanceKlass::low_offset);
    switch (atype) {
        // ===== 静态字段偏移量计算，这里略过
        case STATIC_OOP:
            // ...

        // ===== 非静态字段偏移量计算
        case NONSTATIC_OOP:
            if( nonstatic_oop_space_count > 0 ) {
                real_offset = nonstatic_oop_space_offset;
                nonstatic_oop_space_offset += heapOopSize;
                nonstatic_oop_space_count -= 1;
            } else {
                real_offset = next_nonstatic_oop_offset;
                next_nonstatic_oop_offset += heapOopSize;
            }
            // ...

        case NONSTATIC_DOUBLE:
            real_offset = next_nonstatic_double_offset;
            next_nonstatic_double_offset += BytesPerLong;
            break;
        default:
            ShouldNotReachHere();
    }
    fields->short_at_put(i + instanceKlass::low_offset,
extract_low_short_from_int(real_offset));
    fields->short_at_put(i + instanceKlass::high_offset,
extract_high_short_from_int(real_offset));
}
```

在这段逻辑里面，HotSpot 对 Java 类中的所有字段进行遍历，并分别计算其各个字段的偏移量。以 Java 类中的 long 类型字段为例，其计算逻辑是：

```
case NONSTATIC_DOUBLE:
    real_offset = next_nonstatic_double_offset;
    next_nonstatic_double_offset += BytesPerLong;
    break;
```

`real_offset` 就是当前字段的真实偏移量。假设 Java 类中包含 2 个 `long` 类型的字段，并假设 Hotspot 当前遍历到第一个 `long` 字段，则该 `long` 字段的偏移量就是 `next_nonstatic_double_offset`。

计算完第一个 `long` 字段的偏移量之后，Hotspot 执行 `next_nonstatic_double_offset += BytesPerLong`，将 `next_nonstatic_double_offset` 地址往后偏移 8 字节，这样当 Hotspot 遍历到 Java 类中第 2 个 `long` 类型的字段时，通过 `real_offset = next_nonstatic_double_offset` 就能直接计算出第二个 `long` 字段的偏移量了。

4. gap 填充

也许有细心的小伙伴可能发现上面那段代码中有部分逻辑比较不同寻常，那就是 `case NONSTATIC_DOUBLE` 分支里的逻辑与其他 `case` 分支的逻辑都不一样，其他 `case` 分支的逻辑明显比 `case NONSTATIC_DOUBLE` 这个条件分支的逻辑复杂一些。这是为什么呢？

要解决这个问题，不得不再去关注 JVM 内部的 oop 对象头的事儿。前文也讲到过，每一个 Java 类在堆内存中，都是从 oop 头开始的，而这个 oop 头所占的内存空间与 JVM 是否开启了指针压缩策略有关。在 64 位平台上，如果开启了指针压缩策略，则对象头仅会占用 12 个内存单元，如果没有开启，则会占用 16 个内存单元。

如果 oop 对象头只占用了 12 个内存单元，就会带来一个问题：对象头作为一个整体，不是 8 字节对齐的。对象头作为一个整体是否按 8 字节对齐，与对象头本身没有关系，但是却影响其后面的 Java 类字段的自然对齐效果。由于 JVM 按照 `longs/doubles -> ints -> shorts/chars -> bytes` 的顺序分配内存空间，因此紧跟在 oop 对象头之后的就是 `longs/doubles` 类型的数据。

如果 oop 对象头只占用了 12 字节，那么其后面第一个 `long` 类型的字段的起始偏移量按照常理应该是 12，但这不符合 `long` 类型数据的自然对齐原则（12 不能被 8 整除），所以只能在 oop 对象头后面连续填充 4 个空字节，然后从第 16 个偏移位置处为第一个 `long` 类型的字段分配内存。这样一来就造成从 oop 对象头到第一个 `long` 类型字段之间浪费了 4 字节的内存空间。虽然 4 字节看起来微不足道，但是这对于一个高性能的虚拟机而言是绝对不能接受的。

对内存吝啬到前无古人的地步的 JVM 来说，即使这么一点内存也必须要充分利用起来，而利用的方式便是，将 `int`、`short`、`byte` 等宽度小于等于 4 字节的字段往这个内存间隙里插入，虽然这会破坏 Hotspot 对不同类型字段重排的顺序策略。

我们来看 HotSpot 的源码：

```
清单：/src/share/vm/classfile/classFileParser.cpp
```

```
功能：ClassFileParser::parseClassFile()
```

```
//填充 gap 间隙。如果对象头+父类非静态字段的末尾不是 8 字节(long)对齐，则在这中间填充 Java 类中非 long/double 类型的字段
```

```

//long/double 字段的起始位置还是 8 字节对齐
//gap 填充的顺序: int, short, byte, oopmap
if( nonstatic_double_count > 0 ) {
    int offset = next_nonstatic_double_offset;
    next_nonstatic_double_offset = align_size_up(offset, BytesPerLong);
    if( compact_fields && offset != next_nonstatic_double_offset ) {
        // Allocate available fields into the gap before double field.
        int length = next_nonstatic_double_offset - offset;
        assert(length == BytesPerInt, "");

        //先将 int 型字段填充进 gap, 理论上只能填充 1 个 int, 因为 gap 的总大小最大只能是 7
        nonstatic_word_space_offset = offset;
        if( nonstatic_word_count > 0 ) {
            nonstatic_word_count      -= 1;
            nonstatic_word_space_count = 1; // Only one will fit
            length -= BytesPerInt;
            offset += BytesPerInt;
        }

        //如果填充了 1 个 int 型字段, gap 还没填充完, 则接着填充 short。由于 Java 类中可能并没有定义 int 类型的字段, 因此可以填充多个 short
        nonstatic_short_space_offset = offset;
        while( length >= BytesPerShort && nonstatic_short_count > 0 ) {
            nonstatic_short_count      -= 1;
            nonstatic_short_space_count += 1;
            length -= BytesPerShort;
            offset += BytesPerShort;
        }

        //如果填充完 short 字段之后, 还有 gap 空间, 则继续填充 byte 类型的字段
        nonstatic_byte_space_offset = offset;
        while( length > 0 && nonstatic_byte_count > 0 ) {
            nonstatic_byte_count      -= 1;
            nonstatic_byte_space_count += 1;
            length -= 1;
        }

        //如果 byte 类型的字段填充完了还有 gap 空间, 则继续填充 oopmap
        // Allocate oop field in the gap if there are no other fields for that.
        nonstatic_oop_space_offset = offset;
        if( length >= heapOopSize && nonstatic_oop_count > 0 &&
            allocation_style != 0 ) { // when oop fields not first
            nonstatic_oop_count      -= 1;
            nonstatic_oop_space_count = 1; // Only one will fit
            length -= heapOopSize;
            offset += heapOopSize;
        }
    }
}

```



```

    }
}

```

事实上,如果一个 Java 类显式继承了父类,那么如果父类字段的末尾不是按 8 字节对齐的,则父类字段末尾与子类第一个 long/double 字段之间也会形成补白空隙,则这段空隙也会参与上述逻辑计算,被安插 int、short、byte 等宽度比较小的字段。下文会继续讲解。

5. Java 语言与其他语言处理内存对齐的差异

纵观 HotSpot 对 Java 类字段的堆内存分配算法,可以看出 HotSpot 对内存的利用几乎已经到了极致,虽然存在模仿,但几乎已经无法再被超越了。

做高手的感觉真的好寂寞啊!

我仿佛听到了 Hotspot 内心深处的一声叹息。

别的就不说了,就拿经典的 gcc 编译器来说,最多也只做到了自动将数据类型进行自然对齐,基本不需要开发者再去为这件事情而烦恼,但是也仅此而已。例如前面所列举过的结构体的例子:

清单: 示例程序

作用: C 语言的 struct 内存对齐

```

struct A
{
    int a;
    char b;
    short c;
};

```

struct A 结构体由于内存对齐的需要,最终需要分配 8 字节的内存空间。如果将 A 结构体内部的成员项的顺序变动一下,变成如下:

清单: 示例程序

作用: C 语言的 struct 内存对齐

```

struct B
{
    char b;
    int a;
    short c;
};

```

则内存占用空间立马变大。

这种情况在 Java 中是不存在的。与 struct A 所对等的 Java 类如下:

清单：示例程序

作用：Java 语言的内存对齐

```
class A{  
    int a;  
    byte b;  
    short c;  
};
```

除去 Java 类在 JVM 内部所对应的 oop 的对象头部分的内存空间，最终类 A 字段在堆内存中也会占用 8 字节。

对于 Java 类，不管其内部字段的声明顺序如何变化，都不会影响其内存占用，这主要得益于 JVM 的字段重排算法。

其实经典的编译器诸如 gcc 等，也是可以像 Hotspot 那样，对局部变量进行字段重排的，这在算法层面完全是可行的，只是囿于当时的算法技术而未如此优化。不过事物发展的规律总是长江后浪推前浪，短暂的几十年的历史罅隙，对算法而言可谓是悠久长远的历史长河，再过几十年，说不定会有新的算法横空出世，睥睨天下。不过不管未来怎样，至少从目前来看，JVM 的这种内存分配算法几乎是已经到了无可再优化的境界。

6.2.3 Java 字段内存分配总结

前面浓墨重彩地详细剖析了 Hotspot 对 Java 类字段的内存分配原理，并举了若干例子。这部分内容实在是重要之极，也是理解 JVM 内存模型的最基础、最核心的一步。便连作者本人以前也对 JVM 的内存模型存在诸多误会，以为全是面向对象，以为面向对象的东西必然会浪费非常多的内存空间。直到将 Hotspot 的类字段分配模型剖析完，才发现压根儿不是那么回事，JVM 对内存空间利用率的要求是非常苛刻的，如果去掉占用 12 字节或 16 字节的对象头，其内存使用效率一定超过绝大多数的编程语言了。

本节再对 JVM 的类字段分配策略进行梳理归纳。虽然上面所分析的源码皆基于 Hotspot，但是 HotSpot 所努力实现的目标，事实上正是 JVM 的规范要求。

- ◎ 规则 1：任何对象都是以 8 字节为粒度进行对齐的。
- ◎ 规则 2：类属性按照如下优先级进行排列：长整型和双精度类型；整型和浮点型；字符和短整型；字节类型和布尔类型；最后是引用类型。这些属性都按照各自类型宽度对齐。
- ◎ 规则 3：不同类继承关系中的成员不能混合排列。首先按照规则 2 处理父类中的成员，接着才是子类的成员。

- ◎ 规则 4: 当父类最后一个属性和子类第一个属性之间间隔不足 4 字节时, 必须扩展到 4 字节的基本单位。
- ◎ 规则 5: 如果子类第一个成员是一个双精度或长整型, 并且父类没有用完 8 字节 (没有显式的父类, 并且 JVM 开启了指针压缩策略, oop 对象头只占用 12 字节时), JVM 会破坏规则 2, 按整型 (int)、短整型 (short)、字节型 (byte)、引用类型 (reference) 的顺序向未填满的空间填充。

对于规则 1, 没啥好说的, 与 C 语言中对结构体整体对齐的约束一样, JVM 也需要使 Java 类在 JVM 内部的堆内存映像从整体上做到对齐, 这并不是为了方便 Java 类自己, 而是为了方便其后续的其他类的内存分配。

对于规则 2, 是内存分配时最基本的要求, 自然对齐, 否则 JVM 运行在某些不支持非对齐内存访问的 CPU 硬件上时会因此而崩溃。

对于规则 3, 其实于内存的利用率而言, 并不是一个必须要遵守的原则, 甚至反而因为遵守了这个原则而导致内存利用率降低。这个原则存在的目的主要是为了内存分析时方便, 尤其是当一个类的继承体系比较深的时候, 如果若干父类与子类的字段都混合组合在一起, 那内存分析人员的情绪一定是崩溃的。

对于规则 4, 也是为了让父类属性集合从整体上做到对齐, 从而方便其后续子类字段在处理对齐时能够尽可能地简单。Hotspot 从源码级别保证了规则 4 的履行:

```
first_nonstatic_field_offset = instanceOopDesc::base_offset_in_bytes() +
                               nonstatic_field_size * heapOopSize;
```

这行代码上文讲过, 意在计算 Java 类字段的整体起始偏移量。这个偏移量也要算上 oop 对象头和父类字段, 而父类字段是按照 heapOopSize 对齐的, heapOopSize 的定义如下:

清单: share/vm/utilities/globalDefinitions.cpp

作用: heapOopSize 定义

```
int heapOopSize = 0;
if (UseCompressedOops) {
    // Size info for oops within java objects is fixed
    heapOopSize = jintSize;
    LogBytesPerHeapOop = LogBytesPerInt;
    LogBitsPerHeapOop = LogBitsPerInt;
    BytesPerHeapOop = BytesPerInt;
    BitsPerHeapOop = BitsPerInt;
} else {
    heapOopSize = oopSize;
    LogBytesPerHeapOop = LogBytesPerWord;
    LogBitsPerHeapOop = LogBitsPerWord;
```

```

        BytesPerHeapOop    = BytesPerWord;
        BitsPerHeapOop     = BitsPerWord;
    }

```

如果开启了指针压缩策略，则其大小是 `jintSize`，而 `jintSize` 的值是 4；如果没有开启指针压缩策略，则其大小是 `oopSize`，`oopSize` 的值是 8。

父类的字段大小是 `nonstatic_field_size`，该值的算法如下：

清单：/src/share/vm/classfile/classFileParser.cpp

功能：ClassFileParser::parseClassFile()

```

    int notaligned_offset;
    if( allocation_style == 0 ) {
        notaligned_offset = next_nonstatic_byte_offset + nonstatic_byte_count;
    } else { // allocation_style == 1
        next_nonstatic_oop_offset = next_nonstatic_byte_offset +
        nonstatic_byte_count;
        if( nonstatic_oop_count > 0 ) {
            next_nonstatic_oop_offset = align_size_up(next_nonstatic_oop_offset,
            heapOopSize);
        }
        notaligned_offset = next_nonstatic_oop_offset + (nonstatic_oop_count *
        heapOopSize);
    }
    next_nonstatic_type_offset = align_size_up(notaligned_offset,
    heapOopSize);

    //这里计算的是 field 所占用的字节数，而非 field 的数量。如果是数量，会命名为 count
    //本字段大小由父类的该字段大小加上本类的字段大小
    nonstatic_field_size = nonstatic_field_size +
    ((next_nonstatic_type_offset
    - first_nonstatic_field_offset)/heapOopSize);

```

在这段逻辑中，先对 Java 类中的 `byte` 类型字段进行补白，对齐至 4 字节或 8 字节。由于在 JVM 分配内存时，排在最末尾的是 `byte` 类型，而前面的 `long`、`int`、`short` 这几种类型的字段之间一定不会存在任何补白（因为 `long` 末尾后面的内存位置一定能够保证 `int` 类型字段是自然对齐的，而 `int` 末尾后面的内存位置也一定能够保证 `short` 类型字段是自然对齐的），所以只要确保最末尾的 `byte` 类型的字段能够按照 4 字节或者 8 字节对齐，则整个 Java 所对应的堆内存也一定是按照 4 字节或者 8 字节对齐的。

所以上面这段逻辑能够确保 `next_nonstatic_type_offset` 最终一定也是按照 4 字节或 8 字节对齐。这直接影响到最终所计算出来的父类的 `nonstatic_field_size` 值。

在这段代码的最后一行表达式中，为了让问题简化，假设父类没有父类，所以最后一行表

达式中的 `nonstatic_field_size` 一开始是 0，此时最后一行表达式退化成下面这行表达式：

```
nonstatic_field_size = ((next_nonstatic_type_offset - first_nonstatic_field_offset)/heapOopSize);
```

若 JVM 没有开启指针压缩选项，则 `heapOopSize` 的值为 8，而 `next_nonstatic_type_offset` 经过补白，已经是按照 8 字节对齐的。没有开启指针压缩选项，则 `oop` 对象头占用 16 字节，`first_nonstatic_field_offset` 的值就是 16，所以这行表达式，所有的 3 个变量值都是 8 的整数倍，所以最终计算出来的也必定是 8 的整数倍。

若 JVM 开启了指针压缩选项，则 `heapOopSize` 的值为 4，`next_nonstatic_type_offset` 经过补白，也按照 4 字节对齐。开启指针压缩选项后，`oop` 对象占用 12 字节，则 `first_nonstatic_field_offset` 的值为 12，所以这行表达式中的 3 个变量都是 4 的整数倍，则最终所计算出来的结果也必定是 4 的整数倍。

所以，无论 JVM 是否开启指针压缩选项，则父类字段所需要占用的内存空间必定是 4 的整数倍。这句话换个说法，就是规则 4。

下面这个例子对于本规则有极强的说服力：

清单：Father.java

功能：测试指针压缩

```
public class Father {
    private byte b1;
}

class Son extends Father{
    byte b2;

    public static void main(String[] args){
        SizeOf sizeOf = new AgentSizeOf();
        Father father = new Son();
        System.out.println("father's size: " + sizeOf.sizeOf(father));
    }
}
```

对于规则 5，其实上文已经详细分析过 Hotspot 的源码实现。当开启了指针压缩选项，`oop` 对象头只需要 12 字节的内存空间，如果 Java 类中定义了 `long` 或 `double` 类型的字段，则 `oop` 对象头与第一个 `long/double` 字段之间会有 4 字节的补白空间。JVM 为了充分使用内存，坚决不浪费宝贵的内存空间，就按照整型（`int`）、短整型（`short`）、字节型（`byte`）、引用类型（`reference`）的顺序往这段空隙中填充。

下面这个 Java 类可以验证这一规则：

清单：Father.Java
功能：测试 gap 机制

```
import net.sf.ehcache.pool.sizeof.AgentSizeOf;
import net.sf.ehcache.pool.sizeof.SizeOf;

import java.util.*;

public class Father {
    private long l;
    private byte b1;
    private byte b2;

    public static void main(String[] args){
        SizeOf sizeOf = new AgentSizeOf();
        Father father = new Father();
        System.out.println("father's size: " + sizeOf.sizeOf(father));
    }
}
```

开启指针压缩选项，假设 HotSpot 没有填充 oop 对象头后面的空隙 gap，则此时 Father 类的内存布局如图 6.11 所示。

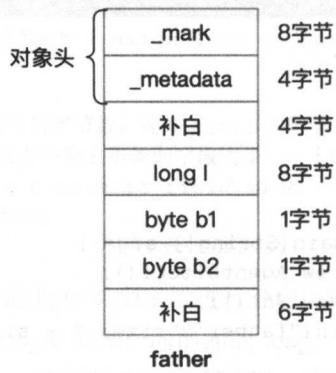


图 6.11 没有 gap 机制时的 Father 类实例内存布局

最终 Father 类在内存中占用 32 字节。而 HotSpot 填充 gap 后，内存布局如图 6.12 所示。



图 6.12 使用 gap 机制后的 Father 类实例内存布局

此时 Father 类在内存中仅占用 24 字节。

而当有父类参与时，则需要同时满足规则 4 与规则 5。下面这个例子正好能够验证这种情况：

清单：Father.java

功能：测试 Java 字段内存对齐

```
import net.sf.ehcache.pool.sizeof.AgentSizeOf;
import net.sf.ehcache.pool.sizeof.SizeOf;

import java.util.*;

public class Father {
    int i;
    short s;
}

class Son extends Father {
    byte b;
    long l;

    public static void main(String[] args) {
        SizeOf sizeOf = new AgentSizeOf();
        Father father = new Son();
        System.out.println("father's size: " + sizeOf.sizeOf(father));
    }
}
```

父类为了满足规则 4，做到按 4 字节对齐，所以父类的末尾补白了 2 个字节，这段空间硬生生被浪费掉了。如果在父类中再定义一个占 2 字节的变量，则 JVM 会用这个字段填充被补白的 2 字节空间。如下面的程序：

清单：Father.java

功能：测试 Java 字段内存对齐

```
import net.sf.ehcache.pool.sizeof.AgentSizeOf;
import net.sf.ehcache.pool.sizeof.SizeOf;

import java.util.*;

public class Father {
    int i;
    short s;
    short s2;

    public static void main(String[] args){
        SizeOf sizeOf = new AgentSizeOf();
        Father father = new Father();
        System.out.println("father's size: " + sizeOf.sizeOf(father));
    }
}

class Son extends Father{
    byte b;
    long l;

    public static void main(String[] args){
        SizeOf sizeOf = new AgentSizeOf();
        Father father = new Son();
        System.out.println("father's size: " + sizeOf.sizeOf(father));
    }
}
```

而即使这样，父类字段的末尾与子类的第一个 long 类型字段之间仍然会有 4 字节的补白。由于子类中包含了 1 个 byte 类型的字段，所以 JVM 违反了规则 2，将这个字段插在了这个间隙里，现在还剩下 3 个补白字节。所以如果子类中继续声明 3 个 byte 类型的字段，则 JVM 会将这 3 个字段插入到剩下的 3 个补白字节中，从而使得 Son 类的堆内存大小依然保持不变。

清单：Father.java

功能：测试 Java 字段内存对齐

```
import net.sf.ehcache.pool.sizeof.AgentSizeOf;
import net.sf.ehcache.pool.sizeof.SizeOf;

import java.util.*;

public class Father {
    int i;
    short s;
```

```

short s2;

public static void main(String[] args){
    SizeOf sizeOf = new AgentSizeOf();
    Father father = new Father();
    System.out.println("father's size: " + sizeOf.sizeOf(father));
}
}

class Son extends Father{
    byte b;
    long l;
    byte b2;
    byte b3;
    byte b4;

    public static void main(String[] args){
        SizeOf sizeOf = new AgentSizeOf();
        Father father = new Son();
        System.out.println("father's size: " + sizeOf.sizeOf(father));
    }
}

```

6.3 从源码看字段继承

在 Java 类的继承关系与细节上,很多小伙伴可能对有些概念存在疑问或误解,下面就从源码的角度逐一分析。

6.3.1 字段重排与补白

分析问题总是免不了要做实验,阅读源码使人明白细节,而实验则使人能够从结果直接验证细节。为了分析 Java 类的一些继承问题,需要通过做实验进行验证,但是在做这个实验之前需要做个实验来先验证字段重排与补白。之所以要验证这个课题,是因为在继承的实验中,基本通过观察父类与子类所占用的内存大小来验证字段是否被继承,而父类与子类所占用的内存大小并不严格等价于其所声明的各个成员本身所占用内存的总和,这还受到内存对齐补白的机制制约。

先从一个最简单的实验用例开始,这是一个空的 Java 类:

清单：/Father.java

功能：一个空的 Java 类

```
public class Father {  
  
}
```

这个空的 Java 类到底占用多大内存空间呢？由于 Java 编程语言并没有提供类似于 C/C++ 语言的 `sizeof` 这种可以获取变量或结构体或类型大小的关键字，因此只能借用第三方类库。这里借用 Ehcache 所提供的 `SizeOf` 工具类。Ehcache 作为广泛使用的分布式缓存中间件，对空间管理必然十分精细，对每一个 Java 类所占用的内存空间大小计算必须十分准确，这是实现内存精细化管理的关键前提。Ehcache 里提供了实现 `SizeOf` 的多种工具版本，这里选择 `AgentSizeOf`。

准备就绪，实验开始，在 `Father` 类中编写 `main()` 函数计算该类所占用的内存大小：

清单：/Father.java

功能：一个空的 Java 类

```
import net.sf.ehcache.pool.sizeof.AgentSizeOf;  
import net.sf.ehcache.pool.sizeof.SizeOf;  
  
public class Father {  
  
    public static void main(String[] args){  
        SizeOf sizeof = new AgentSizeOf();  
        Father father = new Father();  
        System.out.println("father's size: " + sizeof.sizeOf(father));  
    }  
  
}
```

运行 `main()` 函数后打印如下结果（在 64 位平台上，后面的例子都基于 64 位）：

```
father's size: 16
```

这个结果可能出乎很多小伙伴的预料，但是这个结果是十分“伟光正”的。

前面讲到，Hotspot 内部会使用 `oop` 来表示每一个 Java 类的实例，Java 类实例在堆内存中的布局如图 6.13 所示。

`Father` 类没有显式的父类，并且本身是个空类，没有任何私有域，因此该类在堆内存中只有对象头。不过前文讲过，在 64 位平台上，如果开启压缩选项，则对象头占 12 字节的内存空间，否则占用 16 字节。

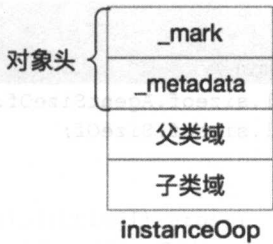


图 6.13 instanceOop 的内存布局

为了确认是否开启压缩选项，使用如下命令查看 JVM 启动的参数：

```
java -XX:+PrintCommandLineFlags
-XX:MaxNewSize=174485504
-XX:MaxTenuringThreshold=4
-XX:NewRatio=7
-XX:NewSize=21811200
-XX:OldPLABSize=16
-XX:OldSize=65433600
-XX:+PrintCommandLineFlags
-XX:+UseCompressedOops
-XX:+UseConcMarkSweepGC
-XX:+UseParNewGC
```

可以看到，JVM 默认开启了压缩参数-XX:+UseCompressedOops。既然使用了压缩算法，则对象头只应该占用 12 字节的内存空间呀，为何 Father 类却占用了 16 字节呢？

这主要是由对齐引起的。JVM 在 64 位平台上为 Java 类对象实例分配内存时，会基于 8 字节的整数倍数进行对齐。如果内存空间不足 8 字节的整数倍，则会将其补白到 8 字节的整数倍。这就是 Father 这个空类占用 16 字节内存的原因。其内存布局如图 6.14 所示。



图 6.14 Father 类的堆内存布局

为了验证这一点，对 Father 类进行改造。先增加一个字节类型的变量：

清单：/Father.java

功能：包含 1 个 byte 类型字段的 Java 类

```
import net.sf.ehcache.pool.sizeof.AgentSizeOf;
import net.sf.ehcache.pool.sizeof.SizeOf;

public class Father {
    private byte b1;

    public static void main(String[] args){
        SizeOf sizeOf = new AgentSizeOf();
        Father father = new Father();
        System.out.println("father's size: " + sizeOf.sizeOf(father));
    }
}
```

现在增加了一个字节类型的变量 `b1`。在 JVM 规范中，一个字节类型的变量仅占 1 字节空间大小。

由于 `Father` 类没有显式继承父类，因此在堆中，其对象头之后应该紧跟着 `b1` 变量。当 JVM 最终为对象头和 `b1` 变量分配内存空间时，由于对象头加上 `b1` 所占用的内存空间总和只有 13 字节，因此 JVM 会将其补白到 16 字节，所以 `Father` 类最终应该仍然占用 16 字节的内存空间。

运行 `main()` 函数，得到结果的确是 16：

```
father's size: 16
```

`Father` 类的堆内存布局如图 6.15 所示。



图 6.15 `Father` 类的堆内存布局

为了继续验证，在 `Father` 类中继续定义 3 个 byte 类型的变量，如下：

清单：/Father.java

功能：包含 4 个 byte 类型字段的 Java 类

```
import net.sf.ehcache.pool.sizeof.AgentSizeOf;
import net.sf.ehcache.pool.sizeof.SizeOf;
```

```

public class Father {
    private byte b1;
    private byte b2;
    private byte b3;
    private byte b4;

    public static void main(String[] args){
        SizeOf sizeOf = new AgentSizeOf();
        Father father = new Father();
        System.out.println("father's size: " + sizeOf.sizeOf(father));
    }
}

```

运行 main()函数，得到如下结果：

```
father's size: 16
```

由于现在对象头加上 4 个 byte 类型的变量的内存总和正好等于 16 字节，正好是 8 的整数倍，因此 JVM 不会对其进行补白。

此时 Father 类的堆内存布局如图 6.16 所示。



图 6.16 Father 类的堆内存布局

接着见证奇迹的时候到了，再增加一个 byte 类型的字节，如下：

清单：/Father.java

功能：包含 5 个 byte 类型字段的 Java 类

```

import net.sf.ehcache.pool.sizeof.AgentSizeOf;
import net.sf.ehcache.pool.sizeof.SizeOf;

public class Father {
    private byte b1;
    private byte b2;

```

```
private byte b3;
private byte b4;
private byte b5;

public static void main(String[] args){
    SizeOf sizeOf = new AgentSizeOf();
    Father father = new Father();
    System.out.println("father's size: " + sizeOf.sizeOf(father));
}
}
```

现在执行 main()函数，得到结果如下：

father's size: 24

此时 Father 类的堆内存布局如图 6.17 所示。

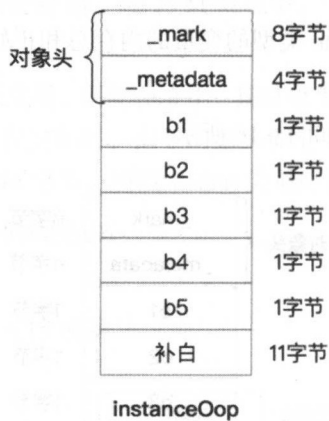


图 6.17 Father 类的堆内存布局

这几个例子已经能够验证 Java 类在内存分配时的对齐机制了。不过 byte 类型的变量不够通用，因此再对 Father 类进行微小的变动，相信小伙伴们能够准确计算出下面这个 Java 类所占的内存空间大小：

清单：/Father.java

功能：包含 2 个 int 类型的 Java 类

```
import net.sf.ehcache.pool.sizeof.AgentSizeOf;
import net.sf.ehcache.pool.sizeof.SizeOf;

public class Father {
    private int i1;
    private int i2;
```



```
public static void main(String[] args){
    SizeOf sizeOf = new AgentSizeOf();
    Father father = new Father();
    System.out.println("father's size: " + sizeOf.sizeOf(father));
}
}
```

现在在 Father 类里面定义了 2 个 int 类型的基本类型变量，JVM 标准规定，一个 int 类型的变量占用 4 字节内存大小。因此 Father 类需要的内存空间大小为：

12 字节(对象头大小)+ 4 字节 * 2(2 个 int 类型变量)= 20 字节

由于 JVM 的内存补白机制，因此最终为其分配 24 字节的内存空间，后面的 4 字节通过补白进行对齐。其内存布局如图 6.18 所示。

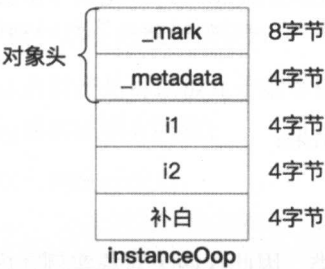


图 6.18 Father 类的堆内存布局

运行 main()函数，的确打印出 24。

关于 JVM 内存对齐的机制就讲到这里，这块理清楚了，有助于接下来要讲的继承机制。

6.3.2 private 字段可被继承吗

有一种说法是，凡是父类中被定义成 private 的字段，都是“老子”的私有财产，即便是“儿子”，也继承不了。

可是 JVM 的世界真的如此无情吗？

看下面这个例子：

清单：/Father.java

功能：演示被 `private` 关键字修饰的字段的继承机制

```
import net.sf.ehcache.pool.sizeof.AgentSizeOf;
import net.sf.ehcache.pool.sizeof.SizeOf;

public class Father {

}

class Son extends Father{
    public static void main(String[] args){
        Father father = new Father();
        Son son = new Son();

        SizeOf sizeof = new AgentSizeOf();

        System.out.println("father's size: " + sizeof.sizeOf(father));
        System.out.println("son's size: " + sizeof.sizeOf(son));
    }
}
```

运行 `main()` 函数，打印如下结果：

```
father's size: 16
son's size: 16
```

本例中的父类与子类都是空类，因此这两个对象实例在内存中其实都是只有对象头，只需要 12 字节的内存空间，但是为了对齐，最终 JVM 为其分配了 16 字节的内存大小。此时父类与子类的内存布局如图 6.19 所示。

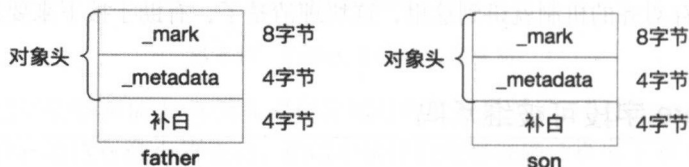


图 6.19 Father 与 Son 实例的堆内存布局

虽然通过上面的例子可以证明，Java 父类中的 `private` 字段的确被子类继承了，证明在 JVM 的世界里其实也是有爱的。可是很多小伙伴对一件事情很是耿耿于怀，因为从编程的角度看，子类并不能直接访问父类中的 `private` 字段。例如下面的例子：

清单: /Father.java

功能: 演示被 `private` 关键字修饰的字段的继承机制

```
public class Father {
    private int b1;
    private int b2;
}
```

```
class Son extends Father{
```

```
    public Son(){
        this.b1;
        super.b1;
    }
}
```

在本例中, 父类中定义了变量 `b1`, 并使用 `private` 关键字进行修饰。在子类 `Son` 的构造函数中, 无论使用 `this.b1`, 还是 `super.b1`, 均会报编译错误。

前面不是证明了“老子”的私有财产是可以被儿子继承的吗, 可是儿子为何却偏偏使用不了老子的私有财产呢? 会不会是前面的证明有问题?

为了弄清楚这个问题, 下面对示例进行改造:

清单: /Father.java

功能: 演示被 `private` 关键字修饰的字段的继承机制

```
import net.sf.ehcache.pool.sizeof.AgentSizeOf;
import net.sf.ehcache.pool.sizeof.SizeOf;
```

```
public class Father {
    private int b1;
    private int b2;
```

```
    public Father(){
        this.b1 = 1;
        this.b2 = 2;
    }
}
```

```
    public int getB2() {
        return b2;
    }
```

```
    public void setB2(int b2) {
        this.b2 = b2;
    }
}
```

```
    public int getB1() {
```

```

        return b1;
    }
    public void setB1(int b1) {
        this.b1 = b1;
    }
}

class Son extends Father{
    public Son(){
        System.out.println("b1 = " + this.getB1());
        System.out.println("b2 = " + this.getB2());
    }

    public static void main(String[] args){
        Son son = new Son();
    }
}

```

运行 main()函数，打印如下结果：

```

b1 = 1
b2 = 2

```

通过本例可以看出，虽然子类不能直接访问父类的私有成员变量，但是却可以通过调用父类为私有成员变量所提供的公开接口访问父类的私有字段。

在 main()函数中执行 Son 的构造函数时，首先要执行父类的构造函数，这是众所周知的。当父类的构造函数执行完成之后，父类的 2 个字段便有了值，所以在子类中能够将父类的 2 个变量值打印出来。但是所谓父类的 2 个字段，其实已经在子类的内存空间里，因此执行父类的构造函数，其实是在初始化子类中所继承的父类部分的字段。

最终所分配的内存模型如图 6.20 所示。

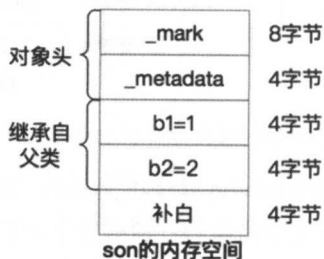


图 6.20 son 的内存空间布局

所以，关于 `private` 关键字的继承问题，应该可以使用如下 2 句话加以概括：

- ◎ “老子”的私有财产（特指私有成员变量）的确是被儿子继承的
- ◎ “儿子”虽然能够继承老子的私有财产，但是却没有权利直接支配，除非老子给儿子开放了接口，否则儿子不能动老子的私有财产。

6.3.3 使用 HSDB 验证字段分配与继承

使用 HSDB 验证 Java 字段的继承：父类 `private` 字段是否继承，父类 `final` 字段是否继承，父类 `private/public/protected` 字段是否被覆盖（引用类型与基本类型）。

前面讲了很多关于一个 Java 类字段大小、字段排序以及在继承的情况下字段覆盖的问题，并且使用工具类来测试了类的大小，从侧面验证了相关理论。但是 JDK 为大家提供了一个神器，其能够直接观察处于运行时的一个 Java 类真实的内存布局以及父类字段的继承与覆盖。

下面就与各位道友一起，通过 HSDB 来直接观察 Java 类在运行期的内存布局。

首先要有测试程序，示例如下，先定义一个父类 `MyClass`：

```
public abstract class MyClass {
    private Integer i = 1;

    protected long plong = 12L;

    protected final short s = 6;

    public char c = 'A';
}
```

接着定义一个子类继承自 `MyClass`：

```
public class Test extends MyClass{
    private long l;
    private Integer i = 3;
    private long plong = 18L;
    public char c = 'B';

    public void add(int a, int b){
        Test test = this;
        int z = a + b;
        int x = 3;
    }

    public static void main(String[] args){
        Test test = new Test();
    }
}
```

```
        test.add(2, 3);
    }
}
```

验证的过程主要使用 HSDB 工具,该工具为 JDK 自带,是一套视窗系统,能够在视窗里面通过界面或者命令行查看运行过程中的若干数据,包含堆栈、堆、perm (JDK8 已经没有 perm 区的概念)、实例数据、常量池,以及与实例所对应的 JVM 内部类 instanceKlass 等。工具的使用也很简单,启动 Java 程序,设置断点后,使用 HSDB 连上 Java 进程即可进行观察。

对于本示例,在 add()函数的 Test test = this 这一句代码上打上断点(可以使用 jdb 命令打断点调试,也可以使用 IDE 集成化的工具),然后使用 HSDB 连接。

本示例使用 jdb 进行调试,首先编译示例程序,得到 Test.class,接着进入 Test.class 所在目录,运行下面命令:

```
jdb -XX:+UseSerialGC -Xmx10m -XX:-UseCompressedOops
```

若在 64 位机器上运行 jdb,为了能够正常使用 HSDB,必须加上-XX:-UseCompressedOops 命令,该命令取消 JVM 的指针压缩功能。如果不加上该选项,则 HSDB 无法正确获取运行期各种数据的内存地址。

执行 jdb 命令后,接着执行下面几个命令:

```
$ javac Test.java
$ jdb -XX:+UseSerialGC -Xmx10m -XX:-UseCompressedOops
正在初始化 jdb...
> stop in Test.add
正在延迟断点 Test.add。
将在加载类后设置。
> run Test
运行 Test
设置未捕获的 java.lang.Throwable
设置延迟的未捕获的 java.lang.Throwable
>
VM 已启动: 设置延迟的断点 Test.add
```

```
断点命中: "线程=main", Test.add(), 行=16 bci=0
16         Test test = this;
```

```
main[1] next
```

```
>
```

```
已完成的步骤: "线程=main", Test.add(), 行=17 bci=2
```

```
17         int z = a + b;
```

```
main[1]
```

上述过程中主要执行了 `stop in Test.add`、`run Test` 和 `next` 这 3 条 `jdb` 命令。`Stop in Test.add` 表示在 `Test` 类的 `add()` 方法处设断点；接着执行 `run Test`，表示开始启动 `Test` 类的 `main()` 函数，`Test` 的 `main()` 函数运行后，`jdb` 会在 `Test.add()` 方法的第一行代码上暂停，此时执行 `next` 命令，于是程序进入 `Test.add()` 方法的第 2 句代码。

`Test` 类的断点调试就到这里，接着执行 `jps` 命令，得到 `Test` 这个 Java 进程的进程 ID，然后启动 `HSDB`，单击 `HSDB` 的 `File->Attach to hotspot process` 菜单选项，输入 `Test` 进程号，即可进入 `HSDB` 界面。

刚进入 `HSDB` 时的界面如图 6.21 所示。

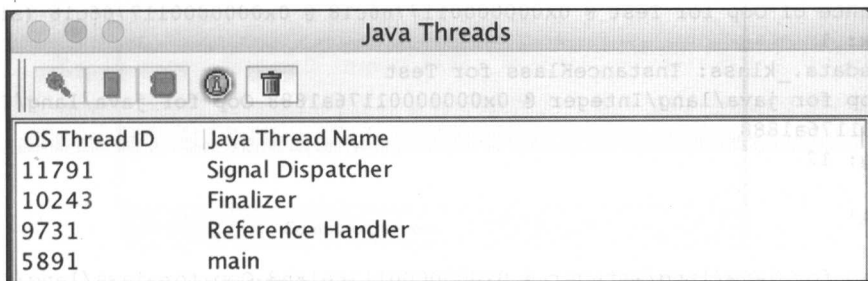


图 6.21 HSDB 连接上 Java 进程后的主窗口

接着单击 `HSDB` 的 `Windows->Console` 菜单选项，打开 `HSDB` 的控制台，刚打开的控制台一片空白，按下回车键，就会出现 `hsdb>` 提示符。接着输入 `universe` 命令，查看当前 Java 进程的堆内存，如下所示：

```
hsdb> universe
Heap Parameters:
Gen 0:   eden [0x0000000011760000,0x00000000117770a90,0x000000001178b0000) space
capacity = 2818048, 53.58432503633721 used
        from [0x000000001178b0000,0x000000001178b0000,0x00000000117900000) space
capacity = 327680, 0.0 used
        to   [0x00000000117900000,0x00000000117900000,0x00000000117950000) space
capacity = 327680, 0.0 used
Invocations: 0

Gen 1:   old [0x00000000117950000,0x00000000117950000,0x00000000118000000) space
capacity = 7012352, 0.0 used
Invocations: 0
```

由于该程序运行于 `JDK 8` 之上，因此并没有显示 `perm space`。如果是 `JDK 6`，则会显示。这里需要注意，虽然 `JDK 6` 与 `JDK 8` 的内存模型有一些重大不同，但是内部类模型并没有根本上的改变，因此使用 `JDK 8` 来运行程序，一样能够完成演示和验证。

接着输入 `scanoops` 命令，在内存中搜索 `Test` 类实例，命令如下：


```
hsdb> scansoops 0x0000000117600000 0x0000000118000000 Test  
0x0000000117766c18 Test
```

`scansoops` 命令接受 3 个参数，分别是搜索的起始地址、终止地址及要搜索的类实例名称。这里所输入的搜索起始地址与终止地址，分别是上面使用 `universe` 命令所计算出的 Gen 0 的起始地址和 Gen 1 的结束地址。

如果能够搜索到类实例对象，`scansoops` 命令会显示该实例在 JVM 内部对应的 `instanceOop` 的内存首地址。果然这里搜索到了一个 `Test` 类的实例地址，该地址是 `0x0000000117766c18`。可以使用 `inspect` 命令查看这个地址处的 oop 的全部数据，如下：

```
hsdb> inspect 0x0000000117766c18  
instance of Oop for Test @ 0x0000000117766c18 @ 0x0000000117766c18 (size = 72)  
_mark: 1  
_metadata._klass: InstanceKlass for Test  
i: Oop for java/lang/Integer @ 0x00000001176a1888 Oop for java/lang/Integer @  
0x00000001176a1888  
plong: 12  
s: 6  
c: 'A'  
l: 0  
i: Oop for java/lang/Integer @ 0x00000001176a18b8 Oop for java/lang/Integer @  
0x00000001176a18b8  
plong: 18  
c: 'B'
```

在 JDK 8 中，Java 类在 JVM 内部所对应的 oop 对象的结构仍然是对象头后面跟着一群类成员变量。除了可以在 HSDB 的 console 命令行中使用 `inspect` 命令查看 oop 对象外，也可以使用图形化的 `Inspect` 界面来观察。单击 HSDB 的 `Tools->Inspector` 菜单选项，输入地址即可，如图 6.22 所示。

图 6.22 所展示的内存布局与前面使用 `inspect` 命令所得到的内存布局是完全相同的。从图中可以看到，`Test` 类所对应的 oop 对象头后面一共跟了 8 个字段，其中对象头与这 8 个字段如图 6.23 所示。

`Test` 类中其实仅定义了 4 个字段，但是其对应的 JVM 内部 oop 对象头后面却跟了 8 个字段，很显然，另外 4 个就是 `Test` 类的父类 `MyClass` 中的字段。父类与子类字段的划分如图 6.24 所示。

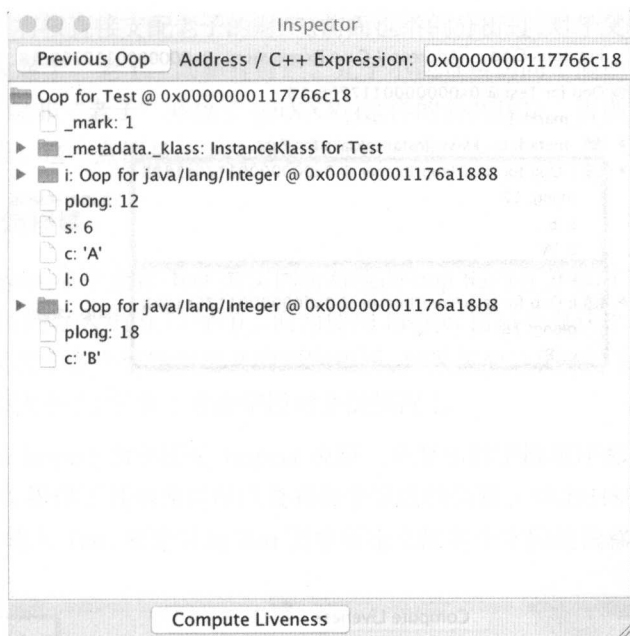


图 6.22 使用 HSDB 的 Inspector 工具查看 instanceClassOop 结构

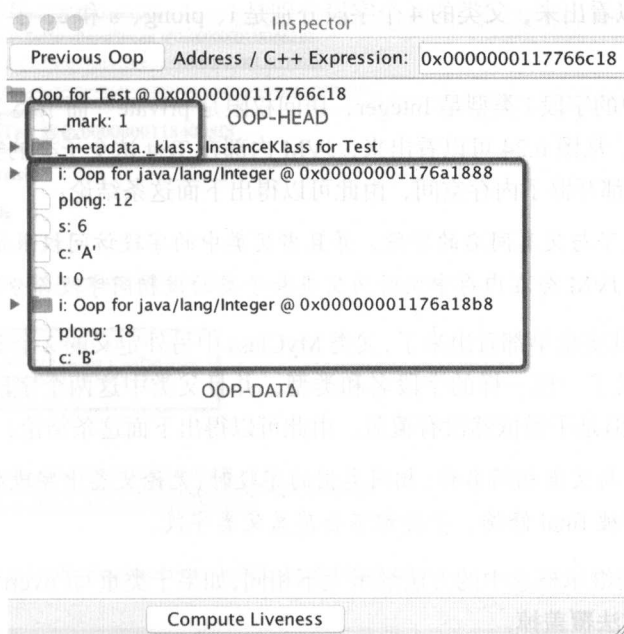


图 6.23 oop 对象头与对象头后面的字段

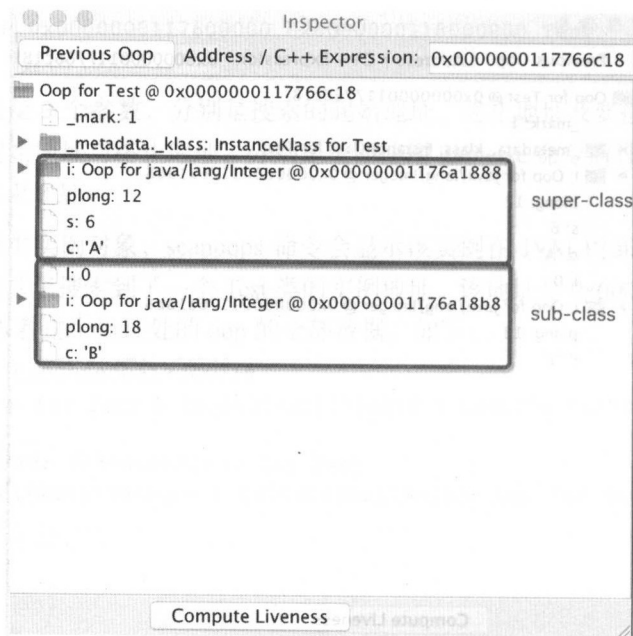


图 6.24 Test 类所对应的 oop 中父类与子类字段的划分

从图 6.24 中可以看出，父类的 4 个字段分别是 i、plong、s 和 c，子类的 4 个字段分别是 l、i、plong 和 c。

父类 MyClass 中的字段 i 类型是 Integer，访问权限是 private。而子类 Test 也定义了一个同名、同类型的字段 i，从图 6.24 可以看出，JVM 内部在 Test 这个子类的实例对象中，同时为父类和子类的变量 i 都开辟了内存空间，由此可以得出下面这条结论：

如果子类定义中与父类同名的字段，并且当父类中的字段访问权限是 private 时，子类不会覆盖父类的字段，JVM 会在内存中同时为父类和子类的该相同字段各分配一段内存空间。

同样，各位道友其实也早都看出来了，父类 MyClass 中另外定义的 2 个变量——plong 和 c，子类 Test 中也都定义了一模一样的字段名和类型，并且父类中这两个字段的访问权限分别是 protected 与 public，但是子类依然没有覆盖。由此可以得出下面这条结论：

当子类中定义了与父类相同名称、相同类型的字段时，无论父类中字段的访问权限是什么，也无论父类字段是否被 final 修饰，子类都不会覆盖父类字段。

这一点与 Java 类继承概念中的方法继承大不相同，如果子类重写(override)了父类的方法，则子类会将父类的方法覆盖掉。

虽然子类的字段不会覆盖父类字段，这意味着“儿子”会全盘接纳“老子”的全部财产，

但是并不等于儿子就有权直接支配老子的财产。前面也举例分析过,对于父类中被定义为 `private` 的字段,这部分字段属于“老子”的私有财产,儿子不能直接访问(例如,儿子不能直接通过 `super.xxx` 来使用),除非“老子”开放了 `getXXX()` 这样的公共接口,否则儿子无论如何都访问不了老子的私有字段。这一点倒是与方法的继承如出一辙。

类成员变量的偏移量

在上面使用 `inspect` 命令查看 `Test` 类实例所对应的 `oop` 的内存分布时(注意,不是 `Inspect` 视窗),显示了 `oop` 的内存大小为 72 字节。而当使用 `Inspect` 窗口查看这个 `Test` 类的实例时,字段的显示顺序却与父类中各个字段所定义的顺序相同,如果按照这种顺序来分配内存,那么 `Test` 类的实例大小可能会大于 72 字节(考虑字段对齐的情况)。

实际上,无论是 `inspect` 命令还是 `Inspect` 视窗,所显示的字段顺序都不是内存中真正分配的顺序。但是 `HSDB` 提供了其他窗口可以查看各个字段的位置,单击 `HSDB` 工具栏的 `Tools->Class Browser` 命令,输入 `Test`,便能看到 `Test` 类中所定义的各个字段的偏移量,如图 6.25 所示。

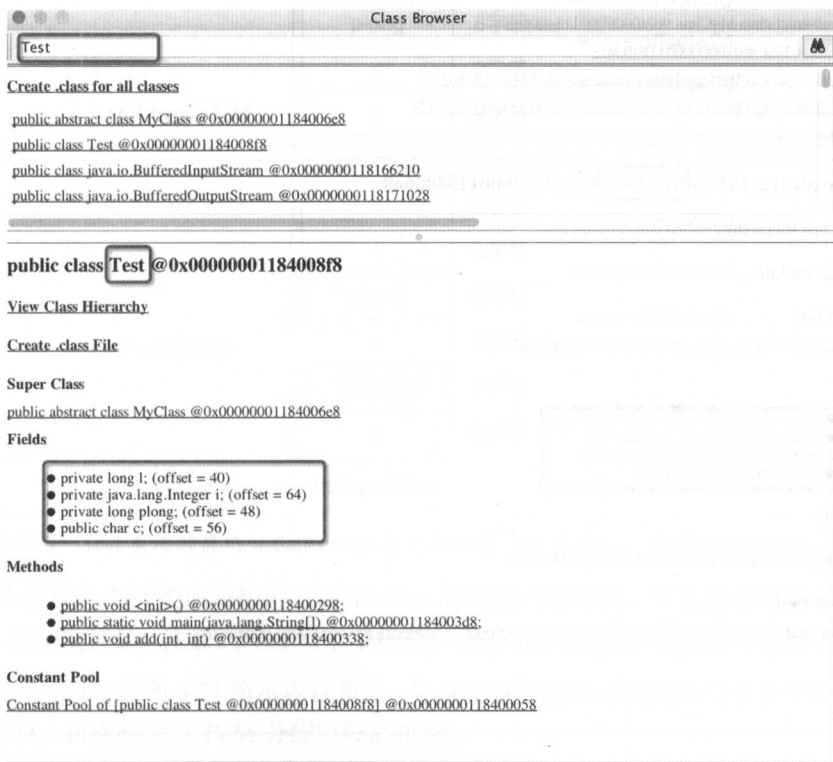


图 6.25 Test 类的字段偏移

图 6.25 显示出 Test 类中所定义的 4 个字段，并且显示出每个字段的偏移量。各个字段的偏移量如下（按照由小到大排列）：

```
private long l:      offset = 40
private long plong:  offset = 48
public char c:       offset = 56
private java.lang.Integer i:      offset = 64
```

可以看出，偏移量最小的字段 l，其偏移量也为 40，而 Test 在 JVM 内部所对应的 oop 的对象头在 64 位机器上最大也仅占用 16 字节，很显然，这是因为在 Test 类自己的字段域与 oop 对象头之间还存在其他数据，而这些数据正是 Test 父类 MyClass 的字段域。

图 6.25 显示了 Test 类的 Super Class，单击 Test 类的父类 MyClass，就显示出 MyClass 类的全部信息，如图 6.26 所示。

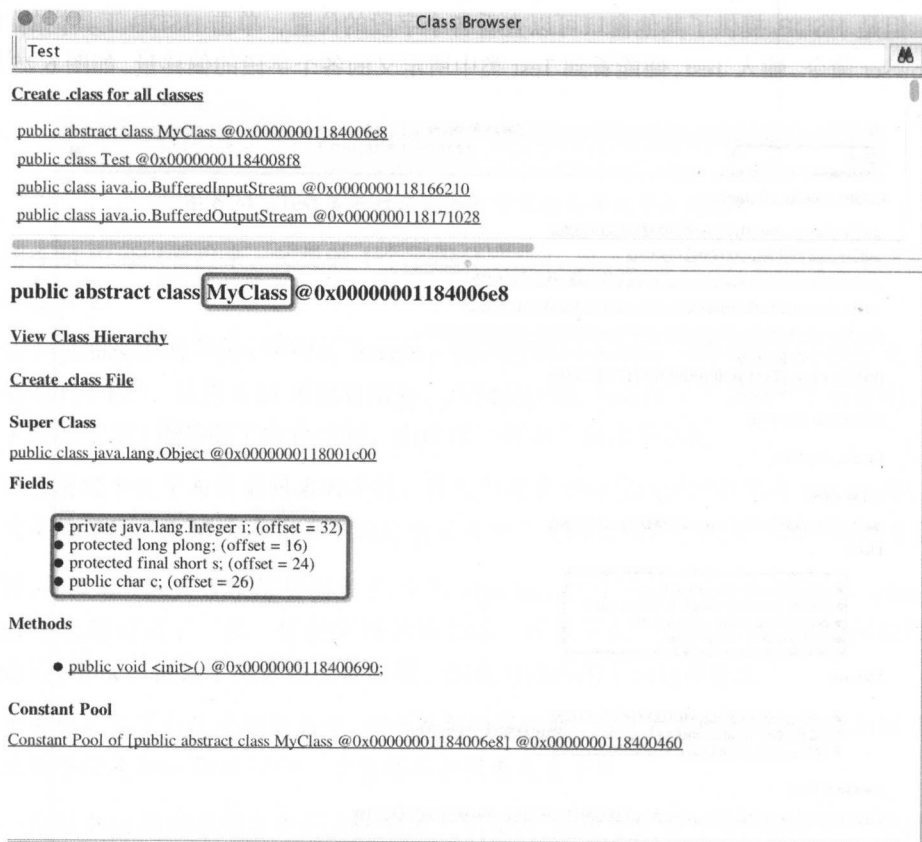


图 6.26 Test 类的父类 MyClass 的字段偏移量

图 6.26 显示了父类 MyClass 中的全部字段及各个字段的偏移量。各个字段的偏移量按照由小到大的顺序分别如下：

```
protected long plong:    offset = 16
protected final short s:  offset = 24
public char c:          offset = 26
private java.lang.Integer i:  offset = 32
```

偏移量最小的字段是 plong，其偏移量是 16，这正好位于 Test 类在 JVM 内部的 oop 对象的对象头之后。而偏移量最大的字段是变量 i，其偏移量是 32。Test 类在 JVM 内部的字段域被分配在其父类字段域之后，而父类 MyClass 的字段域的最后一个字段 i 的偏移量是 32，因此 Test 类的第一个字段 l 的偏移量是 40。

Test 类所对应的 oop 的完整内存布局如图 6.27 所示。

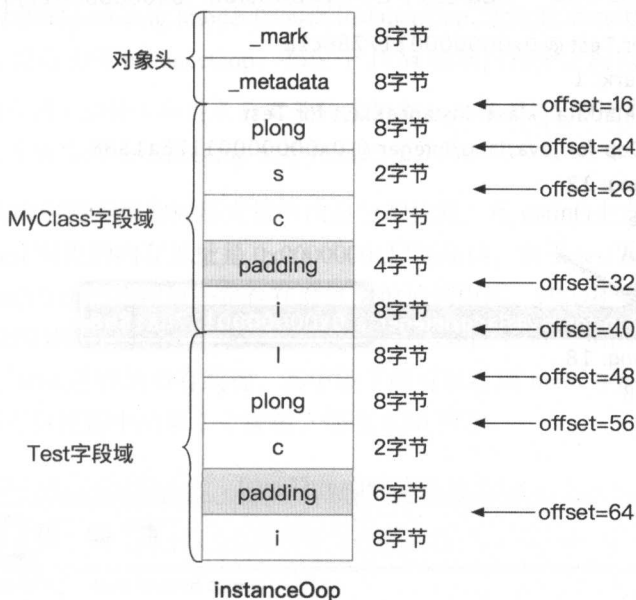


图 6.27 Test 类所对应的 instanceOop 的内存布局（64 位平台，关闭指针压缩选项）

图 6.27 显示出 Test 类所对应的 instanceOop 的完整内存布局，并且显示出各个字段所占的内存大小、起始偏移量。注意，JVM 为了字段对齐，在 instanceOop 里面有两处做了对齐补白。

各位道友可以对照图 6.27 所示内存布局，与 HSDB 中所显示的父类与子类中各个字段的偏移量做一比较，再次感受下 JVM 分配字段的机制。

6.3.4 引用类型变量内存分配

在上面 Test 类的示例中,在 Test 类中定义了引用类型的类成员变量,即 `private Integer i = 3`。同时,在 `main()` 主函数中实例化了 Test 类,得到实例 `test`。变量 `i` 与 `main()` 中的局部变量 `test`,一个是类的成员变量,一个是 Java 方法内的局部变量,但是两者都是引用类型。一起来围观引用类型在两种不同场景下的内存分配吧。

首先看类成员变量 `i`。变量 `i` 既然是 Test 类的成员变量,其应该也在 Test 类实例所对应的 `instanceOop` 的字段域之中,使用 HSDB 的 Inspect 视窗查看在 `main()` 主函数中所创建的 Test 类实例,如图 6.28 所示。

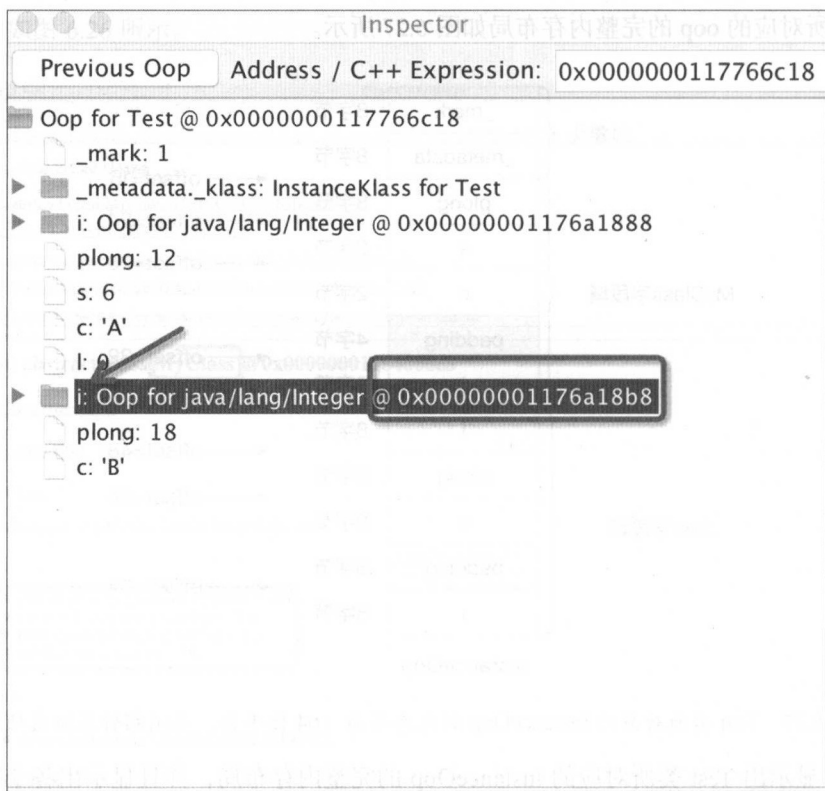


图 6.28 Test 类实例中的成员变量 `i`

图 6.28 显示出 Test 类的成员变量 `i` 的位置和值,其值指向一个 `java/lang/Integer` 类型所对应的 oop,该 oop 地址是 `0x00000001176a18b8`。再使用 HSDB 的 Inspect 视窗查看 Integer 的这个实例地址,如图 6.29 所示。

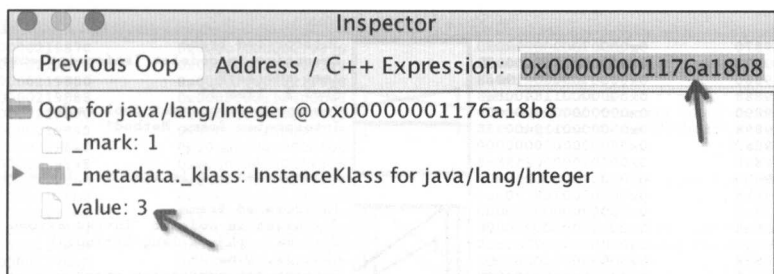


图 6.29 Integer 实例的内存分布

由此可知, Test 类中的成员变量 `i` 的实际数值被存储在 `java.lang.Integer` 类型实例在 JVM 内部所对应的 `instanceOop` 的字段域之中, 而在 Test 类实例所对应的 `instanceOop` 中的成员变量 `i` 所存储的仅仅是一个指针引用。Test 类的成员变量 `i` 的指针引用存储在 Test 类实例所对应的 `instanceOop` 中, `i` 指针指向 `java.lang.Integer` 的实例 `instanceOop`。无论是 Test 类的实例 `instanceOop` 还是 `java.lang.Integer` 类的实例 `instanceOop`, 都位于 JVM 的堆内存中。所以可以得出结论:

类的成员变量的引用(指针)和类成员变量的实例都分配在 JVM 的堆内存中, 类的成员变量的引用分配在所在类的实例 `instanceOop` 的字段域之中。

接着看 Java 方法内引用类型的局部变量的内存分配位置。在 `main()` 主方法里实例化了一个 Test 类的对象 `test`, `test` 对象的内存地址是 `0x0000000117766c18`, 由于 `test` 属于 `main()` 方法的局部变量, 因此在 `main()` 方法的栈帧里一定存在对这个地址的引用。HSDB 支持查看一个线程的整体堆栈内存, 使用 HSDB 刚刚连接上 Java 进程时, 会显示如图 6.30 所示的 Java Thread 窗口, 该窗口主要显示当前 Java 进程的所有线程, 其中最下面可以看到 `main` 主线程, 选中该线程, 单击该窗口上方一排工具按钮中的第 2 个按钮, 如图 6.30 所示。

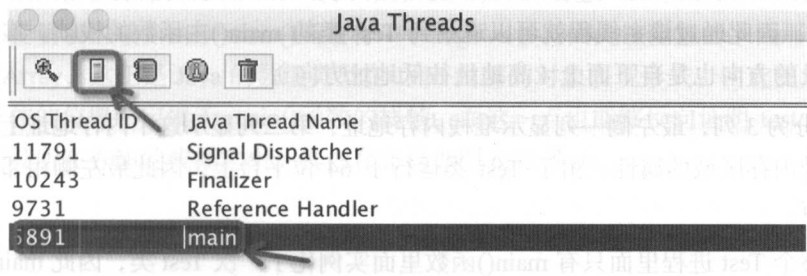


图 6.30 选中 Java 主线程并查看线程堆栈

单击第 2 个工具按钮后, 会弹出新的窗口显示该线程详细的堆栈内存, 如图 6.31 所示。

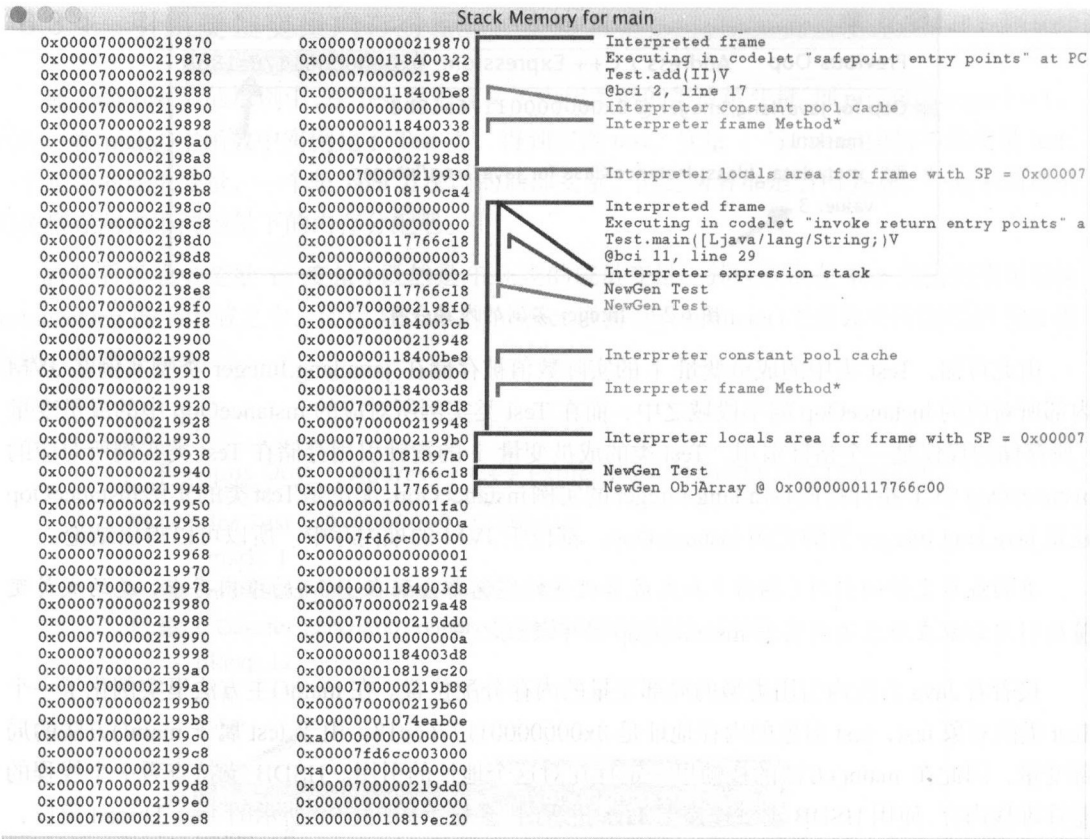


图 6.31 HSDB 显示线程堆栈详情

图 6.31 里显示出当前 Test 进程的 main 主线程, 由于 Java 的主线程最终会调用 Java 程序的 main() 主函数, 因此通过该主线程就可以观察到 Test 类的 main() 主函数的栈帧。堆栈由下往上看, 堆栈增长的方向也是自下而上 (高地址往低地址方向)。

图 6.31 分为 3 列, 最左侧一列显示堆栈内存地址, 第二列显示这个内存地址上的数据, 最右侧显示关键内存区域的属性。由于 Test 类运行于 64 位平台上, 因此最左侧相邻行之间的地址相差 8 字节。

由于在整个 Test 进程里面只有 main() 函数里面实例化了一次 Test 类, 因此 main() 函数的堆栈必然会第一个引用 Test 类实例的地址, 因此只需要在 main() 的栈帧里搜索 test 这个实例的地址。仔细观察这张 main 主线程的堆栈图, 自下而上搜索 Test 类实例的地址, 该地址是 0x0000000117766c18。果然能够找到, 如图 6.32 所示。



图 6.32 寻找 main 主线程堆栈所引用的 Test 类实例

由于这个位置是第一个对 Test 实例对象地址的引用位置，因此该位置一定属于 main() 主函数的局部变量表区域。由于 main() 函数有一个 args 入参，因此该位置的下面那个位置的数据类型是 ObjArray，这正是 Java 的数组类型在 JVM 内部的对象表现形式，由此更加确定图 6.32 中方框所框住的位置一定属于 main() 函数的栈帧，而这一点也能够反向证明 test 这个局部变量的引用位于其所在方法的栈帧之中。其内存布局如图 6.33 所示。

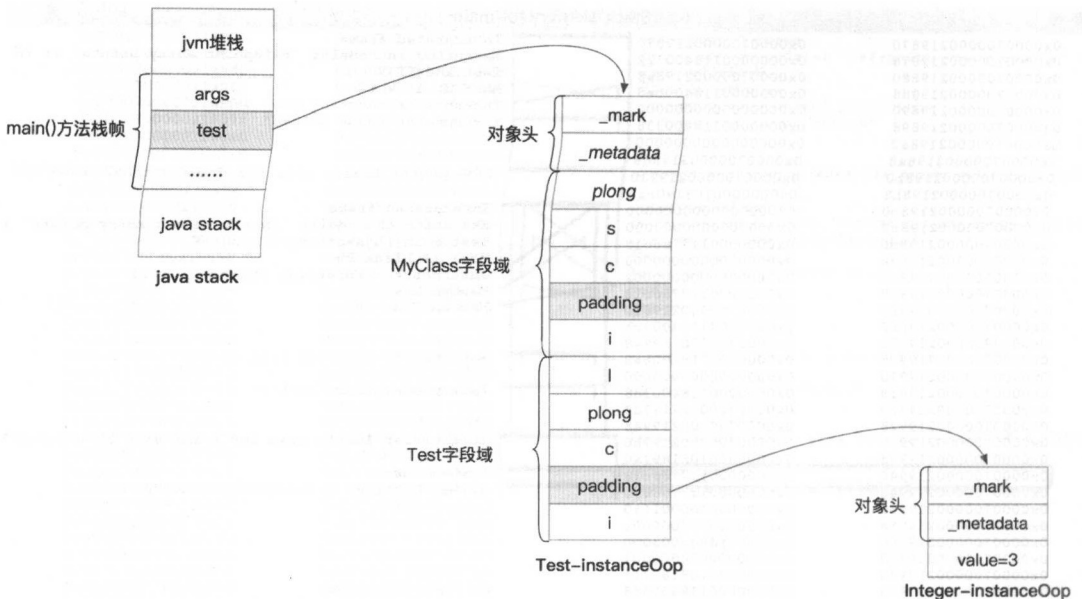


图 6.33 main()函数中的 Test 实例及 Test 类引用成员变量的内存布局

由此可以确定，类成员变量的引用都被分配在堆内存中。

6.4 本章总结

总体而言，HotSpot 解析 Java 类变量的脉络比较清晰，但是也可以看出花了很多心思，这导致 JVM 虽然在执行引擎上相比于那些直接编译成本地机器码的编程语言可能要稍逊一筹，但是在对象的内存分配上，并不比这些编程语言多浪费一点空间（除了每个 Java 类对象必须保留一个对象头），甚至由于字段重排的优化策略，对内存的利用率还要高于这些编程语言的编译器的分配算法。假如分别使用 C++类和 Java 类去描述一个客观事物，当类中包含各种类型的字段，从字节到双精度类型全有，并且字段被乱序声明时，则 Java 类所占用的堆内存空间经过 JVM 的优化之后，甚至会比 C++类所占用的堆内存空间要少。

Java 类在堆内存中的内存空间，主要由 Java 类非静态字段占据。HotSpot 解析 Java 类非静态字段和分配堆内存空间的主要逻辑总结为如下几步：

- (1) 解析常量池，统计 Java 类中非静态字段的总数量，按照 5 大类型（oops、longs/doubles、ints、shorts/chars、bytes）分别统计。
- (2) 计算 Java 类字段的起始偏移量，起始偏移位置从父类继承的字段域的末尾开始。

(3) 按照分配策略, 计算 5 大类型中的每一个类型的起始偏移量。

(4) 以 5 大类型各个类型的起始偏移量为基准, 计算每一个大类型下各个具体字段的偏移量。

(5) 计算 Java 类在堆内存中所需要的内存空间。

经过上面 5 步, HotSpot 便能确定一个 Java 类所需要的堆内存空间。当全部解析完 Java 类之后, Java 类的全部字段信息及其偏移量将会保存到 HotSpot 所构建出来的 `instanceKlass` 中, 至此, 一个 Java 类的字段结构信息便全部解析完成。当 Java 程序中使用 `new` 关键字创建 Java 类的实例对象时, HotSpot 便会从 `instanceKlass` 中读取 Java 类所需要的堆内存大小并分配对应的内存空间。

第 7 章

Java 栈帧

本章摘要

- ◎ entry_point 例程
- ◎ 局部变量表创建的机制
- ◎ 堆栈与栈帧的概念
- ◎ JVM 栈帧创建的详细过程
- ◎ slot 大小到底是多大
- ◎ slot 复用
- ◎ 操作数栈复用与深度

前面的章节一直在讲 JVM 解析 Java 字节码文件的原理和机制，相信很多小伙伴看得有点厌倦了，本章就换个口味，从 Java 字节码的“泥潭”中跳出来，看看别处的美好风景。

众所周知，如果 Java 程序运行出现异常，程序会打印出相应的异常堆栈，通过异常堆栈可以知道 Java 方法的调用链路。其实，调用链路是由一个个 Java 方法栈帧所组成，每一个 Java 方法都有一个栈帧，在这一点上，Java 程序与 C/C++ 程序并无任何区别。本章与各位道友一起分析 JVM 内部实现 Java 方法栈帧的机制和技术实现。

在本章开始讲解之前，让我们将目光再次聚焦到第 2 章。在这一章中，以 JVM 调用 Java 程序的 main() 主函数为例，讲解了 CallStub 例程的实现机制。在 JVM 内部，例程就是一个功能性函数。站在宏观的角度看，它就是一种预先设定好的逻辑。至于逻辑的具体实现方式，可以有很多种，从程序员的角度看，既可以用 C 语言实现，也可以用 Delphi，或者其他编程语言实现。entry_point 例程与 CallStub 例程一样，是一段使用 C 编写，最终生成一段对应的汇编的逻辑。

JVM 调用 Java 程序的 main() 主函数, 会经过 CallStub 例程, 但是在 CallStub 例程里仅仅完成了 Java 主函数的参数传递, 并没有开始执行 Java 程序的 main() 主函数的字节码指令, 这是因为 JVM 在准备执行一个 Java 方法的字节码指令之前, 必须先为该方法创建好对应的方法堆栈。对于 Java 主函数而言, 在 CallStub 例程里会调用 entry_point 例程, 在 entry_point 例程里完成主函数的栈帧创建, 找到 Java 主函数所对应的第一个字节码指令并进入执行。在 entry_point 例程里会涉及 JVM 内部的 method 对象, 即 Java 方法在 JVM 内部的表达形式。关于 method 对象将会在下一章讲解。

JVM 内部可以调用各种不同的方法类型, 例如 JNI 本地函数, 或者 Java 里的静态方法, 或者 Java 类的成员方法。调用不同种类的方法, 会触发不同的 entry_point 例程, 所谓 entry_point, 顾名思义, 就是“进入点”, 进入哪里? 当然是目标方法啦。正因为 JVM 在调用目标方法之前, 会先经过 entry_point, 并且 JVM 在执行目标方法的指令之前, 需要先为其创建好相应的方法堆栈, 因此 JVM 选择在 entry_point 例程中完成方法堆栈创建。本章以 Java 主函数堆栈创建为例, 讲解 JVM 创建 Java 栈帧的具体实现技术。先打个预防针, 本章内容难度属于中等偏上哟! 需要你具备一定的汇编基础知识哟! 不过没有相关基础知识的道友也不必太过于担心, 毕竟大家都挺聪明的!

7.1 entry_point 例程生成

与 CallStub 例程一样, entry_point 例程也是在 JVM 启动过程中被创建。事实上, JVM 内部的所有例程都随着 JVM 的启动而创建。

entry_point 例程的总体创建链路如下 (基于 x86 32 位 Linux 平台):

```
java.c: main()
java_md.c: LoadJavaVM()
jni.c: JNI_CreateJavaVM()
Threads.c: create_vm()
init.c: init_globals()
interpreter.cpp: interpreter_init()
templateInterpreter.cpp: initialize()
templateInterpreter_x86_x32.cpp: InterpreterGenerator()
templateInterpreter.cpp: generate_all()
```

与 CallStub 的伟大“征程”一样, 本链路起步于 JVM 的 main() 函数, 一路走到 init_globals() 这个全局数据初始化模块, 然后便与 CallStub 分道扬镳, 进入 TemplateInterpreter::initialize() 流程。initialize() 函数实现如下:

清单：/src/share/vm/interpreter/templateInterpreter.cpp

作用：initialize()

```
void TemplateInterpreter::initialize() {  
    // ...  
    { ResourceMark rm;  
        TraceTime timer("Interpreter generation", TraceStartupTime);  
        int code_size = InterpreterCodeSize;  
        NOT_PRODUCT(code_size *= 4;) // debug uses extra interpreter code space  
        _code = new StubQueue(new InterpreterCodeletInterface, code_size, NULL,  
                               "Interpreter");  
        InterpreterGenerator g(_code);  
        if (PrintInterpreter) print();  
    }  
    //...  
}
```

在 initialize() 函数中执行了 InterpreterGenerator g(_code) 这行代码，创建解释器生成器的实例。

在 Hotspot 内部，存在 3 种解释器，分别是字节码解释器、C++解释器和模板解释器。字节码解释器逐条解释翻译字节码指令，由于使用 C/C++ 这种高级语言执行字节码指令逻辑，因此执行效率比较低下。

模板解释器相比于字节码解释器的高级之处在于，模板解释器将字节码指令直接翻译成了对应的机器指令，这种直接生成的机器指令相比于字节码解释器所对应的 C/C++ 代码经编译后生成的机器指令，显然要高效很多，毕竟是人力纯手工精雕细琢。

由于模板解释器更加高效，因此 JVM 默认的解释器就是模板解释器，当然可以通过启动参数指定其他解释器。

对于 C++解释器和模板解释器而言，都有一个对应的“解释器生成器”，模板解释器对应的生成器是 TemplateInterpreterGenerator。在 TemplateInterpreter::initialize() 函数中执行 InterpreterGenerator g(_code) 时，实际上是在实例化 TemplateInterpreterGenerator 对象。

TemplateInterpreterGenerator 对象的实例化过程，伴随着其构造函数的调用，其构造函数定义如下（基于 x86 的 32 位 Linux 平台）：

清单：/src/cpu/x86/vm/templateInterpreter_x86_32.cpp

作用：InterpreterGenerator() 构造函数

```
InterpreterGenerator::InterpreterGenerator(StubQueue* code)  
    : TemplateInterpreterGenerator(code) {
```

```

    generate_all(); // down here so it can be "virtual"
}

```

在这里调用了 `generate_all()` 函数。`generate_all()` 顾名思义，就是“产生所有”。所有啥呢？其实就是一款解释器运行时所需要的各种例程及入口。对于模板解释器而言，这些例程直接就是生成好的机器指令。

`generate_all()` 中就包含普通 Java 函数调用所对应的 `entry_point` 的入口，且看 `generate_all()` 的定义：

清单：/src/share/vm/interpreter/templateInterpreter.cpp

作用：`generate_all()` 函数

```

void TemplateInterpreterGenerator::generate_all() {
    // ...

    { CodeletMark cm(_masm, "return entry points");
      for (int i = 0; i < Interpreter::number_of_return_entries; i++) {
        Interpreter::_return_entry[i] =
            EntryPoint(
                generate_return_entry_for(itos, i),
                generate_return_entry_for(itos, i),
                //...
                generate_return_entry_for(vtos, i)
            );
      }
    }

    { CodeletMark cm(_masm, "earlyret entry points");
      Interpreter::_earlyret_entry =
          //...
    }

    { CodeletMark cm(_masm, "deoptimization entry points");
      //...
    }

    { CodeletMark cm(_masm, "result handlers for native calls");
      // ...
    }

    //...

#define method_entry(kind) \
    { CodeletMark cm(_masm, "method entry point (kind = " #kind ")"); \
      Interpreter::_entry_table[Interpreter::kind] = \
          generate_method_entry(Interpreter::kind); \
    }

```

```

    }

    // all non-native method kinds
    method_entry(zerolocals)
    method_entry(zerolocals_synchronized)
    method_entry(empty)
    method_entry(accessor)
    //...

    // all native method kinds (must be one contiguous block)
    Interpreter::_native_entry_begin = Interpreter::code()->code_end();
    method_entry(native)
    method_entry(native_synchronized)
    Interpreter::_native_entry_end = Interpreter::code()->code_end();

#undef method_entry

    // Bytecodes
    set_entry_points_for_all_bytes();
    set_safe_points_for_all_bytes();
}

```

对于模板解释器而言，本方法无疑具有里程碑式的意义，它将生成模板解释器所对应的各种模板例程的机器指令，并保存入口地址。如果一个启动的 JVM 代表一个高度进化的文明社会，那么这个方法一定代表着一个国家的重大基础设施建设工程，所有的高速公路、高铁网络、地铁交通、机场与航线、港口与轮渡，等等，都会在本阶段里完工。Java 程序中的一个对象类型如同这个社会里的一个个鲜活的人类个体，基础设施完成以后，社会里的人可以借助于高效快速的各种交通网络去完成各自的神圣使命。

generate_all()函数的上半段定义了一些重要的逻辑入口，例如 CodeletMark cm(_masm, “return entry points”)代码段定义了 return 指令的入口，同时会生成其对应的机器指令。而从 #define method_entry(kind) 宏定义开始，则定义了一系列“方法入口”，例如，zerolocals、abstract、java_lang_math_sin 等。

在 AbstractInterpreter 中定义了 JVM 所支持的全部方法入口：

清单：/src/share/vm/interpreter/AbstractInterpreter.cpp

作用：MethodKind 枚举声明

```

enum MethodKind {
    zerolocals,
    zerolocals_synchronized,
    native,
    native_synchronized,
    empty,

```

```

    accessor,
    abstract,
    method_handle,
    java_lang_math_sin,
    java_lang_math_cos,
    java_lang_math_tan,
    java_lang_math_abs,
    java_lang_math_sqrt,
    java_lang_math_log,
    java_lang_math_log10,
    java_lang_ref_reference_get,
    number_of_method_entries,
    invalid = -1
};

```

当 JVM 调用 Java 函数时，例如 Java 类的构造函数、类成员方法、静态方法、虚方法等，或者特定的数学函数，最终就会从不同的入口进去，在 CallStub 例程中进入不同的函数入口。

对于正常的 Java 方法调用（包括 Java 程序主函数），其所对应的 entry_point 一般都是 zerolocals 或者 zerolocals_synchronized，如果方法加了同步关键字 synchronized，则其 entry_point 是 zerolocals_synchronized。因此这里关注 zerolocals 方法入口，method_entry(zerolocals)生成了该方法入口。method_entry()正是在 generate_all()中定义的宏，调用 method_entry(zerolocals)就相当于执行了下面这个逻辑：

```

Interpreter::_entry_table[Interpreter::zerolocals] =
    generate_method_entry(Interpreter::zerolocalsfd)

```

这个逻辑执行完之后，JVM 会为 zerolocals 生成本地机器指令，同时将这串机器指令的首地址保存到 Interpreter::_entry_table 数组中。这与一个国家统一注册登记各地的高速入口、机场、港口、高铁站等是一样的道理。

对于 32 位 x86 Linux 平台，为 zerolocals 方法入口生成机器指令的 generate_method_entry() 函数定义如下：

清单：/src/cpu/x86/vm/templateInterpreter_x86_32.cpp

作用：generate_method_entry()函数

```

address AbstractInterpreterGenerator::generate_method_entry(AbstractInterp
reter::MethodKind kind) {
    // determine code generation flags
    bool synchronized = false;
    address entry_point = NULL;

    switch (kind) {
        case Interpreter::zerolocals : break;
        case Interpreter::zerolocals_synchronized: synchronized = true; break;

```

```

        case Interpreter::native : entry_point =
            ((InterpreterGenerator*)this)->generate_native
            _entry(false); break;
        case Interpreter::native_synchronized : entry_point =
            ((InterpreterGenerator*)this)->generate_native
            _entry(true); break;
        //...
        case Interpreter::java_lang_ref_reference_get : entry_point =
            ((InterpreterGenerator*)this)->generate_Reference
            _get_entry(); break;
        default : ShouldNotReachHere();
        break;
    }

    if (entry_point) return entry_point;

    return ((InterpreterGenerator*)this)->generate_normal_entry(synchronized);
}

```

在 `generate_method_entry()` 函数中，判断入参的枚举类型，当入参类型是 `zerolocals` 时啥也不干，因此跳出 `switch` 条件判断分支，直接到最后一句：

```
return ((InterpreterGenerator*)this)->generate_normal_entry(synchronized)
```

在 `generate_normal_entry()` 函数中，终于要开始为 `zerolocals` 生成本地机器指令了。在 32 位 x86 Linux 平台上，`generate_normal_entry()` 函数定义如下（为了少占篇幅，以下仅摘录主要逻辑）：

清单：/src/cpu/x86/vm/templateInterpreter_x86_32.cpp

作用：generate_normal_entry() 函数

```

address InterpreterGenerator::generate_normal_entry(bool synchronized) {
    //...
    address entry_point = __ pc();

    //定义寄存器变量
    const Address size_of_parameters(rbx,
    methodOopDesc::size_of_parameters_offset());
    const Address size_of_locals(rbx, methodOopDesc::size_of_locals_offset());
    const Address invocation_counter(rbx,
    methodOopDesc::invocation_counter_offset() +
    InvocationCounter::counter_offset());
    const Address access_flags(rbx, methodOopDesc::access_flags_offset());

    //获取 Java 方法入参数量、max_locals 及局部变量 slot 数量
    __ load_unsigned_short(rcx, size_of_parameters);
    __ load_unsigned_short(rdx, size_of_locals)
    __ subl(rdx, rcx);
}

```

```

//获取返回地址
__ pop(rax);

//计算 Java 方法第一个入参在堆栈中的地址
__ lea(rdi, Address(rsp, rcx, Interpreter::stackElementScale(), -wordSize));

//为局部变量 slot (不包含 Java 方法入参) 分配堆栈空间, 初始化为 0
{
    Label exit, loop;
    __ testl(rdx, rdx);
    __ jcc(Assembler::lessEqual, exit);           // do nothing if rdx <= 0
    __ bind(loop);
    __ push((int32_t)NULL_WORD);                 // initialize local variables
    __ decrement(rdx);                           // until everything initialized
    __ jcc(Assembler::greater, loop);
    __ bind(exit);
}

//创建栈帧
generate_fixed_frame(false);

//引用计数
Label invocation_counter_overflow;
Label profile_method;
Label profile_method_continue;
if (inc_counter) {
    generate_counter_incr(&invocation_counter_overflow, &profile_method,
&profile_method_continue);
    if (ProfileInterpreter) {
        __ bind(profile_method_continue);
    }
}
Label continue_after_compile;
__ bind(continue_after_compile);

//开始执行 Java 方法第一条字节码
#ifdef ASSERT
{ Label L;
    const Address monitor_block_top (rbp,
        frame::interpreter_frame_monitor_block_top_offset *
wordSize);
    __ movptr(rax, monitor_block_top);
    __ cmpptr(rax, rsp);
    __ jcc(Assembler::equal, L);
    __ stop("broken stack frame setup in interpreter");
    __ bind(L);
}

```

```

    }
#endif

    // jvmti support
    __ notify_method_entry();

    //进入 Java 方法第一条字节码
    __ dispatch_next(vtos);

    return entry_point;
}

```

`generate_normal_entry()`函数在 JVM 启动过程中调动，执行完成之后，会向 JVM 的代码缓存区写入对应的本地机器指令。当 JVM 调用一个特定的 Java 方法时，会根据 Java 方法所对应的 `entry_point` 类型找到对应的函数入口，并执行这段预先生成好的机器指令。

通过前面的分析可知，`CallStub` 例程所进入的 `entry_point` 例程其实就是方法入口，并且这个方法入口并不是只有一个，而是有一批。至于到底会进入哪一个入口，其实在编译期就确定了，编译器会判断 Java 方法的签名（Java 方法名、访问标识，是否使用 `synchronized` 锁定，是否是虚方法等），并根据 Java 方法的签名信息生成不同的方法调用指令。在一个 Java 类被 JVM 加载的过程中，同样会对每个 Java 方法进行签名信息分析，并最终确定一个 Java 方法的 `entry_point` 类型。

下面开始具体分析 `generate_normal_entry()`函数执行的详细过程。由于 Java 方法栈主要由局部变量表、帧数据和操作数栈这三大部分组成，因此下面的讲解也主要从这几个方面进行讲解。

7.2 局部变量表创建

7.2.1 `constMethod` 的内存布局

世界是物质的，物质是运动的，运动是有规律的，规律是可以被掌握的。当 JVM 启动运行后，其内部的一切数据对象都处于不断运动的状态，而运动的规律，则由詹爷一手创建。相对于 JVM 内部的所有对象而言，詹爷就是“它们”的创世神。我等后辈如果能够掌握这种运动的规律，那么不是神也成仙了（哈哈）。

JVM 是使用 C/C++写成的，与其他基于 C 的程序一样，JVM 充分基于数据结构展开其独特的算法。JVM 内部变化的是对象的位置，是对象的诞生和消亡，是指向对象的指针。同样，

算法也会一直变化。但是唯一千年不变的是对象所代表的数据结构，是一串结构中各个元素的相对偏移。这种相对位移便成了 JVM 内部物质运动的一种内在规律，连 Java 函数在 JVM 内部所对应的 method 数据结构表达形式也挣脱不了这种造物主所设定的命运。

作为 JVM 世界的造物神，自然是知道这种规律的，因为规矩就是造物神定下的。在 `entry_point()` 例程中，这种规律将被用来定位 Java 函数所对应的字节码位置，并计算局部变量表的容量。对于 JDK 6，JVM 内部通过偏移量为 Java 函数定下了“规矩”，准确地说，至少定下了 3 条规矩：

- ◎ method 对象的 `constMethod` 指针紧跟在 `methodOop` 对象头的后面，也即 `constMethod` 的偏移量是固定的。
- ◎ `constMethod` 内部存储 Java 函数所对应的字节码指令的位置相对于 `constMethod` 起始位的偏移量是固定的。
- ◎ method 对象内部存储 Java 函数的参数数量、局部变量数量的参数的偏移量是固定的。

注：`constMethod` 对象是 `method` 对象内部的一个字段。`method` 对象是 Java 方法在 JVM 内部所对等的数据结构，该结构会在下一章详细讲解。

这种偏移量是由 C++ 编译器保证的。JDK 6 的 `method` 及其相关属性的偏移量如图 7.1 所示。

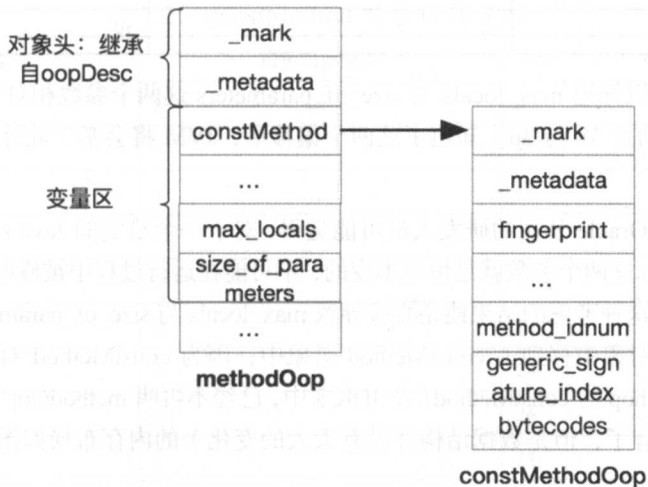


图 7.1 JDK 6 的 `methodOopDesc` 内部布局

首先看 `constMethod`，其相对于 `methodOop` 起始位置的偏移量是一个 `oopDesc` 对象头的距离。在 32 位 x86 平台上，这个距离为 8，即两个指针的宽度。

再看 Java 函数的两个十分重要的属性：max_locals 和 size_of_parameters。在 JDK 6 中，这两个参数被保存在 methodOopDesc 对象中，在 32 位 x86 平台上，methodOopDesc 的成员变量、所占用的内存大小（以字节为单位）、相对于 methodOop 起始位置的偏移量（以字节为单位）如表 7.1 所示。

表 7.1 methodOopDesc 对象内存分配

成 员 变 量	占用内存空间	偏 移 量
header	4	0
klass	4	4
constMethodOop	4	8
constants	4	12
methodData	4	16
interp_invocation_count	4	20
access_flags	4	24
vtable_index	4	28
method_size	2	32
max_stack	2	34
max_locals	2	36
size_of_parameters	2	38

基于表 7.1，可以知道 max_locals 与 size_of_parameters 这两个参数相对于 methodOop 对象首地址的偏移量分别是 38 与 36。知道了这两个偏移量，JVM 将会基于此计算局部变量表的大小。

而到了 JDK 8，Oracle 公司的研发人员可能觉得，对于一个给定的 Java 函数，其 max_locals 与 size_of_parameters 这两个参数就是恒定不变的，不可能在运行过程中被修改，因此应该将其当作只读的属性。基于这种考虑的结果便是直接导致 max_locals 与 size_of_parameters 这两个参数被从 methodOopDesc 对象中移到了 constMethod 对象中，因为 constMethod 对象中的属性都是只读的。JDK 8 中，methop 与 constMethod（在 JDK 8 中，已经不再叫 methodOop 和 constMethodOop 了，后面的 Oop 没有了，但是数据结构并没有太大的变化）的内存布局如图 7.2 所示。

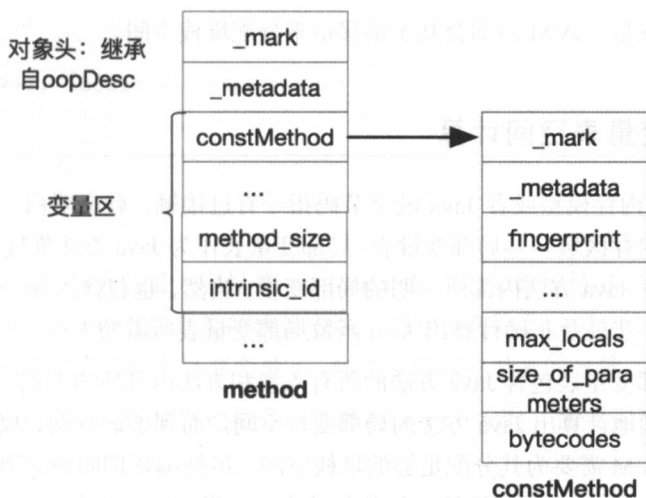


图 7.2 JDK 8 的 method 对象内存布局图

所以，在 JDK 8 中，要想从 methodOop 对象中读取到 max_locals 和 size_of_parameters 这两个参数，便不能再基于 method 的偏移量去读取了，只能基于 constMethod 的偏移量进行读取。在 32 位 x86 平台上，JDK 8 中的 constMethod 类型的成员变量、所占用的内存大小（以字节为单位）、相对于 constMethod 起始位置的偏移量（以字节为单位）如表 7.2 所示。

表 7.2 constMethod 对象内存分配

成员变量	占用内存空间	偏移量
fingerprint	8	0
constants	4	8
stackmap_data	4	12
constMethod_size	4	16
flags	2	20
code_size	2	22
name_index	2	24
signature_index	2	26
method_idnum	2	28
max_stack	2	30
max_locals	2	32
size_of_parameters	2	34

记住这里的偏移量，JVM 内部会基于偏移量来分配堆栈空间。

7.2.2 局部变量表空间计算

如果你对 JVM 内存模型或者 Java 的字节码指令有过接触，就会明白，要研究它们怎么都绕不开一个常见的内存区域——局部变量表。局部变量表作为 Java 方法堆栈(栈帧)的一部分，主要的作用就是保存 Java 方法内部所声明的局部变量，当然，也包含入参。成功为 Java 函数分配局部变量表的第一步就是正确计算出 Java 函数局部变量表所需的大小。

Java 方法的局部变量表包含 Java 方法的所有入参和方法内部所声明的全部局部变量。在编译阶段，编译器准确地计算出 Java 方法的局部变量空间。而到了运行期，仅仅知道局部变量空间大小是不够的，JVM 需要为其分配足够的堆栈空间。虽然编译期间便知道了所需要的局部变量空间大小，但是这对 JVM 进行堆栈空间分配并不能提供足够的信息，因为通常情况下，Java 方法的入参的堆栈空间是由调用方所分配，因此被调用方并不需要再分配编译期所计算出的全部局部变量空间。对于 Java 方法之间的调用（是指一个 Java 方法调用了另一个 Java 方法，而非本地函数调用 Java 方法），调用方的操作数栈与被调用方的局部变量表往往存在重叠区，这给 Java 方法局部变量表的分配带来一定的挑战(栈帧重叠的原因会在后续章节进行详细讲解)。

所以在运行期，JVM 只需要为 Java 方法的局部变量分配堆栈空间，而不需要为 Java 函数的入参额外分配空间，因为入参的堆栈空间由调用方完成分配。

很绕，有没有？

事实上，这么绕是有道理的，根本原因是编译期间所获取的信息与运行期间所能获取的信息是不对称的，并且运行期间可能会有各种优化。

对于绕的东西，举例来理解是一个不错的办法。下面是一个简单的 Java 类，包含一个很简单的方法：

清单：A.java

作用：局部变量表空间示例

```
class A{
    public void add(int x, int y){
        int z = x + y;
    }
}
```

使用 `javap -verbose` 命令进行分析，得到的信息如下：

```
public void add(int, int);
descriptor: (II)V
```

```
flags: ACC_PUBLIC
```

```
Code:
```

```
    stack=2, locals=4, args_size=3
```

```
    0: iload_1
```

```
    1: iload_2
```

```
    2: iadd
```

```
    3: istore_3
```

```
    4: return
```

add()方法的局部变量表的最大容量是 4 (locals=4), 入参数量是 3 (args_size=3)。入参数量之所以是 3, 是因为 add()方法是类的成员方法, 因此会有隐藏的第一个入参 this。3 个入参, 加上 add()方法内部所定义的一个局部变量 z, 构成了局部变量表的容量。

注意看 add()方法所对应的字节码指令, 当执行 $z = x + y$ 时, 需要先执行两条字节码指令:

```
iload_1
```

```
iload_2
```

这两条字节码指令分别将局部变量表的第 1 个槽位和第 2 个槽位的数据推送至表达式栈栈顶 (槽位起始编号从 0 开始)。第 1 和第 2 个槽位上所保存的数据, 正是 add()方法的两个入参 x 和 y。很显然, 第一个槽位上所保存的数据是 this 指针。更加显然的是, JVM 内部的局部变量表的确包含了入参。

当执行完 $z = x + y$ 后, 对应的最后一条字节码指令是 istore_3, 它将计算结果保存到局部变量表的第 3 个槽位。第 3 个槽位正是 add()方法内部所定义的局部变量 z。

此时的局部变量表的内部结构与索引如下:

槽位索引	对应入参/局部变量
0	this
1	x
2	y
3	z

注: 在 32 位平台上, 一个槽位占 32 位。

对于本例, 由于 x 和 y 这两个入参在调用方调用 add()方法时便已经分配完毕, 因此 JVM 无须再为这两个入参分配堆栈空间, 只需为 z 分配。而 z 所需的堆栈空间大小, 则为编译期间所计算出的局部变量表的大小减去入参数量, 对于本例而言, 就是 $(4-3)=1$ 。因此, JVM 只需要为 add()方法内的局部变量 z 分配 1 个变量槽。

entry_point 例程里给出了这种除入参之外所需要的局部变量表的空间大小的计算方法:

清单：/src/cpu/x86/vm/templateInterpreter_x86_32.cpp

作用：generate_normal_entry()函数

```
address InterpreterGenerator::generate_normal_entry(bool synchronized) {
    //...
    const Address size_of_parameters(rbx,
methodOopDesc::size_of_parameters_offset());
    const Address size_of_locals    (rbx,
methodOopDesc::size_of_locals_offset());
    const Address invocation_counter(rbx,
methodOopDesc::invocation_counter_offset() + InvocationCounter::counter_offset());
    const Address access_flags      (rbx, methodOopDesc::access_flags_offset());

    // 获取 Java 函数入参数量
    __ load_unsigned_short(rcx, size_of_parameters);

    // 获取 Java 函数局部变量表最大槽数
    __ load_unsigned_short(rdx, size_of_locals);

    //最大槽数减去 Java 函数入参数量，得到除入参之外所需要分配的堆栈空间大小
    __ subl(rdx, rcx);

    //...
}
```

最终生成的机器指令如下（使用汇编展示，基于 32 位 x86 平台，JDK 6）：

```
movzwl 0x26(%ebx),%ecx
movzwl 0x24(%ebx),%edx
sub    %ecx,%edx
```

从 call_stub 例程进入 entry_point 例程之前，ebx 寄存器指向 Java 函数所对应的 method 对象首地址，根据前文所分析的 JDK 6 中 method 的结构可知，相对于 method 首地址偏移 38 个字节的位置保存的是 max_locals 参数，而偏移 36 个字节的位置保存的是 size_of_parameters 参数。38 和 36 所对应的十六进制正是 0x26 和 0x24。

到了 JDK 8，由于 max_locals 与 size_of_parameters 这两个参数保存的位置发生了变化，因此其寻址方式也跟着发生变化。JDK 8 最终所生成的机器指令如下：

```
mov    0x8(%ebx),%edx
movzwl 0x22(%edx),%ecx
movzwl 0x20(%edx),%edx
sub    %ecx,%edx
```

由于在 JDK 8 中，max_locals 与 size_of_parameters 这两个参数从 method 对象移到了 constMethod 对象中，因此首先执行 mov 0x8(%ebx),%edx，将 edx 寄存器指向 constMethod 对象

首地址。因为 `constMethod` 相对于 `method` 对象的首地址的偏移量为 8 字节,而程序流从 `call_stub` 例程进入 `entry_point` 例程之后, `ebx` 寄存器指向 `method` 对象首地址,因此通过 `0x8(%ebx)` 将 `ebx` 寄存器往上移动 8 字节的宽度,就得到 `constMethod` 首地址,并将这个首地址保存到 `edx` 寄存器中。接着再基于 `constMethod` 的偏移量分别通过 `0x22(%edx)` 与 `0x20(%edx)`,就得到 `max_locals` 与 `size_of_parameters` 这两个参数值。

7.2.3 初始化局部变量区

在 `CallStub` 执行 `call %eax` 指令之前,物理寄存器(注:不是逻辑寄存器哦)中所保存的重要信息如表 7.3 所示。

表 7.3 物理寄存器中的信息

寄存器名	指向
<code>edx</code>	<code>parameters</code> 首地址
<code>ecx</code>	Java 函数入参数量
<code>ebx</code>	指向 Java 函数,即 Java 函数所对应的 <code>method</code> 对象
<code>esi</code>	<code>CallStub</code> 栈顶

计算除 Java 方法入参之外的参数所需要分配的堆栈空间,接着就是执行空间分配。

对于绝大多数 C/C++ 语言,包括 Delphi 等可以直接编译为本地二进制机器指令的语言,分配堆栈空间基本都使用如下指令:

```
sub operand, %esp
```

`esp` 寄存器指向调用者函数的栈顶,要扩展堆栈的内存空间,只需将栈顶指针继续往下移动,移动多大空间,就能扩展多大空间。当然也不是随便想扩充多大就扩充多大,对于基于硬件的操作系统而言,往往有一个最大堆栈深度的限制;而对于基于软件的 JVM 虚拟机,本身也提供了这种限制,具体的限制通过参数 `-Xss` 进行控制。

而在 `entry_point` 例程中,分配堆栈空间使用了另一种方式,那就是进行 `push` 操作。`entry_point` 例程所对应的源码如下:

清单: `/src/cpu/x86/vm/templateInterpreter_x86_32.cpp`

作用: `generate_normal_entry()` 函数

```
address InterpreterGenerator::generate_normal_entry(bool synchronized) {
    // ... return address 即可
    // get return address
```



```

__ pop(rax);

// compute beginning of parameters (rdi)
__ lea(rdi, Address(rsp, rcx, Interpreter::stackElementScale(),
-wordSize));

// rdx - # of additional locals
// allocate space for locals
// explicitly initialize locals
{
    Label exit, loop;
    __ testl(rdx, rdx);
    __ jcc(Assembler::lessEqual, exit);    // do nothing if rdx <= 0
    __ bind(loop);
    __ push((int32_t) NULL_WORD);          // initialize local variables
    __ decrement(rdx);                    // until everything initialized
    __ jcc(Assembler::greater, loop);
    __ bind(exit);
}
//...
}

```

这段逻辑主要执行了 3 件事：

- (1) 将栈顶的返回地址暂存到 `rax` 寄存器中。
- (2) 获取 Java 函数第一个参数在堆栈中的位置。
- (3) 为局部变量表分配堆栈空间。

在 32 位 x86 平台上，这段逻辑所生成的机器指令如下（使用汇编展示）：

```

// get return address
pop    %eax

// compute beginning of parameters (rdi)
lea    -0x4(%esp,%ecx,4),%edi //让 edi 指向第一个参数在堆栈中的位置

// rdx - # of additional locals
// allocate space for locals
// explicitly initialize locals
test   %edx,%edx    //测试 edx 是否为 0，即判断 Java 函数中是否有局部变量
jle    0x01cbbb08    //如果 Java 函数内部没有声明局部变量，则跳过堆栈空间分配
push   $0x0
dec    %edx
jg     0xb36d6559

```

下面对局部变量表的初始化逻辑进行详细讲解。

1. 暂存返回地址

JVM 控制流从 `call_stub` 例程刚进入 `entry_point` 例程时，尚未对堆栈空间进行任何操作，因此到目前为止，`call_stub` 例程与 `entry_point` 例程的堆栈空间如图 7.3 所示。

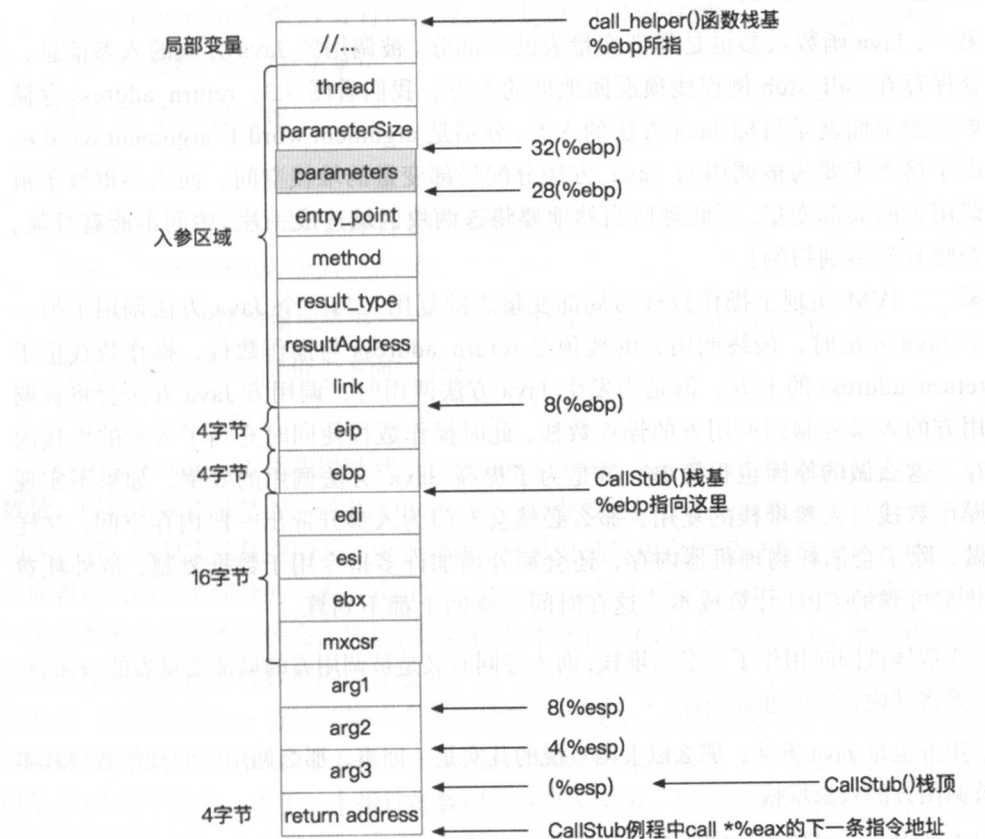


图 7.3 `CallStub` 执行完 `call *%eax` 指令后的内存布局

可以看到，在栈顶位置保存的是目标 Java 函数的返回地址，注意这个所谓“返回地址”的概念，在上一章讲解 `CallStub` 例程时对此进行过专门描述，它是指 `CallStub` 例程中“`call *%eax`”这条指令所在内存区域的下一条指令地址，这个值实际上是原本 `CallStub` 例程中 `eip` 寄存器的值。因此，这里的“返回地址”并不是指目标 Java 函数的返回值。不过要注意的是，JVM 内部也将 `return address` 叫作“`return_from_java`”，因此下文有时根据需要写成 `return_from_java`，诸君心里知道这其实也是 `return address` 即可。不管是 `return address` 也好，还是 `return_from_java` 也罢，其实都是 `eip`。

接下来，JVM 要为被调用者函数的局部变量分配堆栈空间，在分配之前，需要先将“返回地址”临时保存到 `rax` 寄存器中。

但是为何要这样做呢？

原因主要包括以下两方面：

- ◎ 第一，Java 函数入参也是局部变量表的一部分。被调用的 Java 方法的入参信息，就保存在 `call_stub` 例程栈顶返回地址的上方，我们看图 7.3，`return_address` 存储单元上面就是目标 Java 方法的入参，分别是 `argument word 1~argument word n`。由于接下来要为被调用的 Java 方法分配局部变量的堆栈空间，而入参也属于被调用方的局部变量，因此理所当然地要将这两块区域连成一片，中间不能有分隔，否则看着多别扭啊！
- ◎ 第二，JVM 实现了操作数栈与局部变量表的复用。当一个 Java 方法调用了另一个 Java 方法时，最终调用方的栈顶是 `return address` 与操作数栈，操作数栈位于 `return address` 的上方。但是当发生 Java 方法调用时，调用方 Java 方法会将调用方的入参复制到调用方的操作数栈，此时操作数栈便同时充当了入参的堆栈内存。这么做的原因也很简单，就是为了提高 Java 方法调用的效率，如果不实现操作数栈与入参堆栈的复用，那么必然会专门为入参开辟出一段内存空间，这样做，除了会消耗物理机器内存，还会额外增加许多指令用于数据复制，额外耗费非常可观的 CPU 计算成本，这在时间与空间上都不划算。

既然操作数栈被同时用作了入参的堆栈，而入参同时又是被调用方的局部变量表的一部分，因此就必然要将其内存空间连成一片。

如果调用方也是 Java 方法，那么以上两点说的其实是一回事，那么调用方的操作数栈其实同时也是被调用方的入参堆栈。

基于以上两点，入参堆栈与即将分配的局部变量的堆栈之间不允许存在一个碍事的 `return address` 参数，因此 JVM 便将这个参数先移走。

JVM 对 `return address` 说：“麻烦您老兄先到别处凉快会儿，这儿暂时没您啥事儿，等这边的事儿完了，到时再麻烦您老兄移驾复位。”

`return address` 心里想：“我去！没事老拿我开涮！”，嘴上应声道：“好嘞！”

于是 JVM 念起那段古朴沧桑的强大咒语：`pop %eax`，将 `return address` 瞬间传送到 `eax` 中。

2. 获取 Java 函数第一个入参在堆栈中的位置

JVM 念完咒语, return address 瞬间消失, 现在 JVM 从栈顶一眼看过去, 第一个映入眼帘的参数变成了 argument word n , 也就是 Java 方法的最后一个入参。

argument word n 想跟 JVM 打声招呼, 但是 JVM 并没有理睬, 而是大声呼叫:

“第一个参数, 给我站出来!”

喊完了, 但是没人理它。

悲哀!

JVM 一言不合就念咒语, 那古老的咒语是:

```
lea    -0x4(%esp,%ecx,4),%edi
```

念完咒语, CPU 偷偷地将第一个参数的堆栈坐标写进了 edi 寄存器。

将这个咒语展开, 等价于下面这个表达式:

```
(%edi) = (%esp) + (%ecx) * 4 - 0x4
```

程序流从 call_stub 例程跳转到 entry_point 例程时, ecx 寄存器中记录的是 Java 方法入参的数量, 假设入参数量是 N , 则由此可知, 第一个参数的位置在当前栈顶往上 N 字节, 再往下移动 4 字节。这里需要注意的是, 由于堆栈空间从高地址内存位置向低地址内存位置增长, 所以上面表达式的前半部分的子表达式 $(\%esp) + (\%ecx) * 4$ 所计算出的结果实际上是 Java 方法第一个入参的内存位置的最高位地址, 但是在大部分主流 CPU 架构平台上, 都是将一个数据的最低位内存地址标记为该数据的内存地址, 而不是最高位内存地址, 因此上面表达式最后需要减去 0x4, 从而得到 Java 方法的第一个入参的最低位地址, 这就是第一个入参的内存首地址。同时需要注意的是, 在 32 位平台上, 一个指针类型的数据宽度是 4 字节, 因此这里减去的是 4; 但是在 64 位平台上, 由于一个指针的数据宽度是 8 字节, 因此在 64 位平台上, 上面这段表达式就变成如下这样:

```
(%edi) = (%esp) + (%ecx) * 8 - 0x8
```

在该表达式中, 自然数 0x4 变成 0x8。

这个“经”之所以这么念, 是有道理的, JVM 可是精打细算的主! 刚才在赶走 return address 老兄时, 所念的咒语是 pop %eax, 这段咒语念完, 栈顶的数据被弹出, esp 寄存器会自动往上移动, 移动之后的堆栈内存布局如图 7.4 所示。

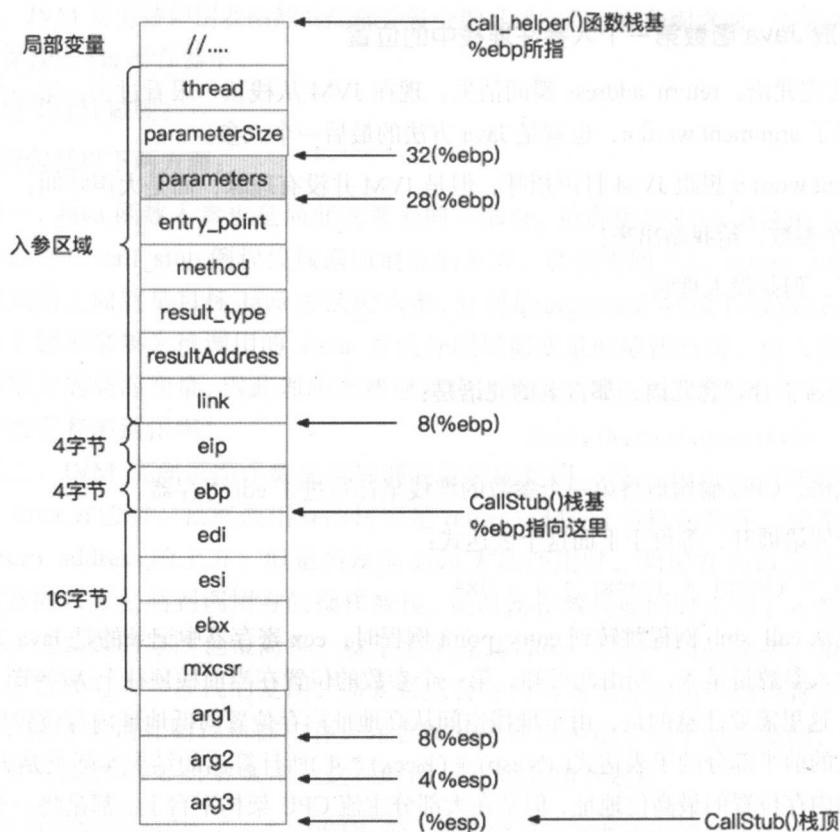


图 7.4 将 return address 弹出之后的堆栈内存布局

由于现在 `rsp` 寄存器指向了最后一个参数，因此需要往上移动 $(n-1) * 4$ 字节的位置，找到第一个参数的堆栈地址。注意，JVM 在这里使用了 `lea` 命令，而非 `mov`。`lea` 命令是专门用于获取内存地址的命令。

在这一步之所以要将 Java 函数的第一个入参的位置保存下来，是因为接下来要为 Java 函数内部所声明的局部变量分配堆栈空间，并且要执行 Java 字节码指令，字节码指令往往都会涉及在操作数栈与局部变量表之间相互传送数据，而 JVM 要读取/写入局部变量表，必然要知道局部变量表的起始位置，否则无法定位。因此在这一步将局部变量表的起始位置保存到 `edi` 寄存器中，后续相关的字节码指令例如 `iload`、`istore` 等都需要用到 `edi` 寄存器，直接从 `edi` 寄存器中读取局部变量表的起始位置。

更进一步说，这也是对局部变量表读取和写入时都基于索引的原因，因为 JVM 的确就是基于局部变量表的起始位置做偏移的，从而读取/写入局部变量表相关数据。局部变量的索引号其

实就是入参或局部变量相对于局部变量表的偏移量。

Java 方法的第一个入参的内存位置将作为 Java 方法的局部变量表的起始位置,并保存在 edi 寄存器中。这里其实偷偷地埋下了一个巨坑,假设一个 Java 方法包含 3 个入参,同时假设该方法没有局部变量,则该方法的局部变量表及 edi 寄存器所指向的位置如图 7.5 所示。

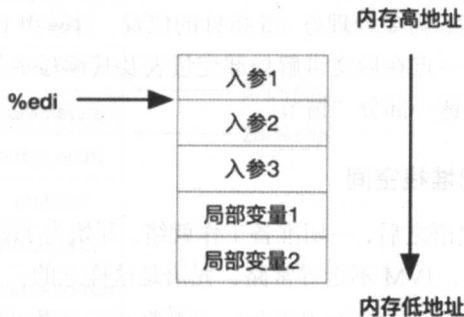


图 7.5 Java 局部变量表的 slot 槽位起始位置

edi 寄存器所保存的正是 Java 方法第一个入参的内存位置,Java 方法的局部变量表的 slot 位置也以该位置为起始点。而事实上,对于计算机内部的一段连续的数据区域,其内存首地址是整个数据区域中地址最低的那个位置,很显然,slot 的起始位置并不符合这一原则。图 7.5 中,由于 Java 方法堆栈由内存高地址向低地址方向增长,因此 Java 方法的第一个入参的内存地址反而处于最高位,因此如果按照一般逻辑,Java 方法的局部变量表的 slot 起始位置应该如图 7.6 所示才对。

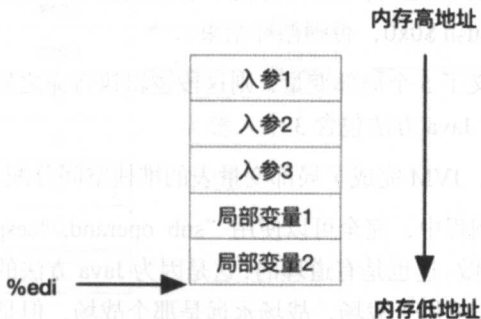


图 7.6 Java 局部变量表的 slot 槽位起始位置——按一般逻辑

由于 JVM 将整个局部变量表的高位地址作为其起始位置,因此运行期的字节码指令操作局部变量表的 slot 槽位索引时,必须充分考虑到这一点,而这也是理解读写局部变量表所对应的

load 与 store 系列的字节码指令所对应的机器码的关键所在，否则将会一头雾水。同时，在进一步计算入参位置时，默认将 Java 方法入参全部当成指针类型，因此在 32 位和 64 位平台上，Java 方法的第一个入参的内存位置要么以 32 位计算，要么以 64 位计算，但是在局部变量表中，普通数据类型（例如 int）与 long 类型所占的槽数不同，而无论在 32 位还是 64 位平台上，局部变量表中的 long 类型的数据宽度都要比指针类型的数据宽度大，因此 JVM 在处理局部变量表中的 long/double 类型的数据时，需要处理好 edi 指针的位置，计算出 long/double 类型数据在局部变量表中的真实偏移量。这一点在后文讲解局部变量表及其读写字节码指令所对应的机器码时会详细讲解。这里先提前剧透一部分“情节”。

3. 为局部变量表分配堆栈空间

JVM 念完两个古老的咒语之后，一切准备工作就绪，开始为 Java 方法内部所声明的局部变量分配堆栈空间了。这一次，JVM 不走寻常路，咒语是这样念的：

```
test    %edx,%edx    //测试 edx 是否为 0，即判断 Java 函数中是否有局部变量
jle     0x01cbbb08    //如果 Java 函数内部没有声明局部变量，则跳过堆栈空间分配
push     $0x0         //0xb36d6559
dec      %edx
jg       0xb36d6559    //这个地址就是上面第 3 行的指令地址，push $0x0
```

这段指令的逻辑是，先测试 edx 是否为 0，edx 寄存器中所保存的结果就是前面 max_locals - size_of_parameters 后得到的值。如果这个值为 0，则说明被调用的 Java 方法内部并没有声明任何局部变量，于是 JVM 不会再去分配堆栈空间。能省一件事就省一件事，整天念咒语，累啊！

如果 max_locals - size_of_parameters 不等于 0，则执行一个微循环，先通过 push \$0x0 往栈顶压入一个 0，接着通过 dec %edx 将 edx 寄存器中的值减去 1，最后再判断 edx 的值是否为 0，如果不是 0 则继续跳转回 push \$0x0，否则循环结束。

假设 Java 方法内部定义了 5 个局部变量，则这段逻辑执行完之后，最后的堆栈内存布局如图 7.7 所示（仍然假设目标 Java 方法包含 3 个入参）。

通过不断 push 的方式，JVM 完成了局部变量表的堆栈空间分配。

其实，在 entry_point 例程中，完全可以使用“sub operand, %esp”这样的方式分配堆栈空间，但是这里偏偏没有这么做。这也是有道理的，这是因为 Java 方法的堆栈空间会被反复使用。对于一片指定的堆栈区域，就像个战场，战场永远是那个战场，但是却前赴后继来了一茬一茬又一茬的军队。战争有它自己的规律，每次完成一场决战，胜利的一方都要打扫战场，清理物资。而对于同一块内存区域，这一次可能是被作为 Java 方法 a() 的堆栈，而下一次则可能变成方法 b() 的堆栈。与战场所不同的是，Java 方法执行完毕之后，并不负责清零堆栈，因此清零的工作只能由下一次使用这块堆栈空间的 Java 方法去负责。这便是在 entry_point 例程中使用循环

push 0 的方式进行分配堆栈空间的原因。如果使用“sub operand, %esp”的方式去分配堆栈空间，分配完了仍然需要进行一次循环，使用“mov 0, -operand(%esp)”这样的方式对堆栈空间进行清零，效率反而低下，不美。

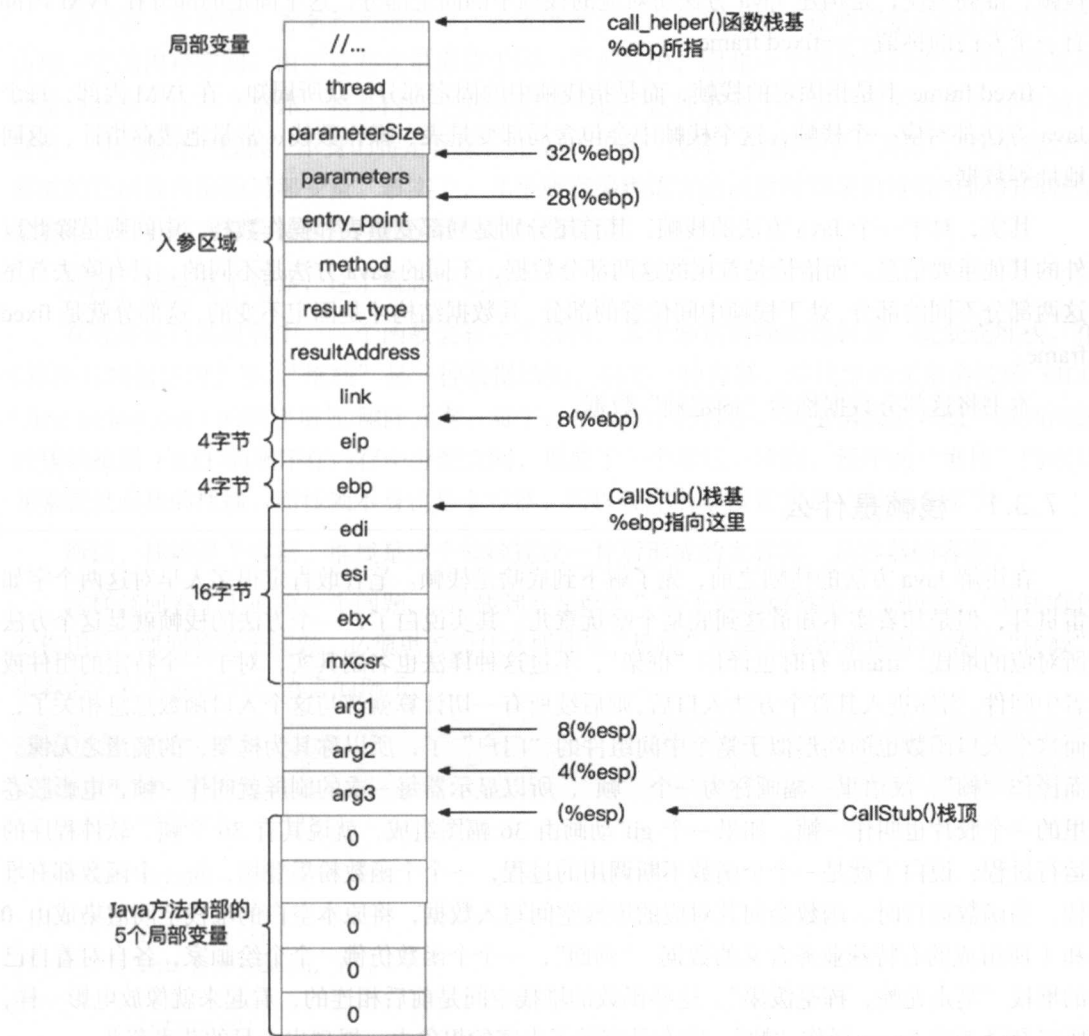


图 7.7 JVM 为局部变量分配堆栈空间

7.3 堆栈与栈帧

JVM 为局部变量分配完堆栈空间后，接着执行一项特殊的任务。这项特殊的任务就是构建栈帧，准确地说，是构建 Java 方法所对应的栈帧中的固定部分，这个固定的部分在 JVM 内部有一个专门的称谓——fixed frame。

fixed frame 不是指固定的栈帧，而是指栈帧中的固定部分。众所周知，在 JVM 内部，每个 Java 方法都对应一个栈帧，这个栈帧中会包含局部变量表、操作数栈、常量池缓存指针、返回地址等数据。

其实，对于一个 Java 方法的栈帧，其首尾分别是局部变量表和操作数栈，中间则是除此以外的其他重要信息。而恰恰是首尾的这两部分数据，不同的 Java 方法是不同的，只有除去首尾这两部分不同的部分，处于栈帧中间位置的部分，其数据结构才是固定不变的，这部分就是 fixed frame。

本书将这部分数据称为“固定帧”数据。

7.3.1 栈帧是什么

在讲解 Java 方法的栈帧之前，先了解下到底啥是栈帧。笔者敢肯定很多人早对这两个字如雷贯耳，但是却着实不知道这到底是个啥玩意儿。其实说白了，一个方法的栈帧就是这个方法所对应的堆栈。frame 有时也译作“框架”，不过这种译法也名副其实，对于一个特定的组件或者中间件，当你进入其首个方法入口后，则后续所有一切计算就都与这个入口函数息息相关了，而这个入口函数也的确形似于整个中间组件的“门户”了，所以称其为框架，的确当之无愧。而译作“帧”，汉语里一幅画称为一个“帧”，所以显示器每一次的刷屏就叫作一帧，电影胶卷里的一个胶片也叫作一帧。如果一个 gif 动画由 36 幅图组成，就说其有 36 个帧。软件程序的运行过程，说白了就是一个个函数不断调用的过程，一个个函数粉墨登场，每一个函数都有堆栈，当函数运行时，函数会向其对应的堆栈空间写入数据，将原本空白的堆栈空间渲染成由 0 和 1 所组成的有特殊业务含义的数据“画面”，一个个函数仿佛一个个画家，各自对着自己的堆栈“笔走龙蛇，挥毫泼墨”。这些函数的堆栈空间是前后相连的，看起来就像放电影一样，所以翻译者将 frame 译作“帧”，实在是充满了丰富的想象力，展现出十足的艺术范儿。

当然，除了栈帧的叫法，很多人也将其叫作“活动记录”，这主要来源于一个英文单词——“activation records”。当然活动记录与堆栈的概念还有一些细微的区别，活动记录更多用于特指当前位于栈顶的堆栈，因为 CPU 只关心最顶端的堆栈。所以关于堆栈的叫法实在是不少，不过现在就让我们忘了这些名字或概念，而是去思考下面两个问题：

(1) 堆栈到底是啥?

(2) 堆栈有什么作用?

要回答这两个问题,还得从机器的角度说起。

对于第一个问题,答案其实很简单,函数内部会定义局部变量,这些变量最终要在内存中占用一定的内存空间。由于这些变量都位于同一个函数中,因此一个很自然的想法就是将这些变量合起来当作一个整体,将其内存空间分配在一起,这样有利于变量空间的整体内存申请和释放。所以抛开“栈帧”本身的特定含义,完全可以将“栈帧”看作一个“容器”,这个容器中存放的是函数内部的局部变量。事实上,几乎所有编程语言的函数所对应的堆栈空间的确就是个容器,在容器内部会按顺序存放函数内的局部变量。

所以,栈帧是个容器,这个概念要记住。

在程序运行的过程中,一个函数会有一个栈帧,多个函数的栈帧连起来,就变成堆栈。在《算法与数据结构》里,“堆栈”是一种数据结构,也是一种容器。堆栈里的元素会按照 FILO (first in/last out) 的顺序增加/删除元素。对于一个运行中的程序,从主函数进入的一系列函数的栈帧按照 FILO 的顺序在内存中分配空间,形成了一个堆栈。所以,程序的“堆栈”的成员元素就是函数的栈帧,而栈帧本身也是个容器,所以程序的堆栈其实是“容器的容器”。

所以,栈帧是个容器,堆栈是多个栈帧连成一片后形成的大容器,是容器的容器。

这样就回答了上面第一个问题,“堆栈到底是啥?”接下来要解答第二个问题,“堆栈有什么作用?”要回答这个问题,必须思考为什么要使用“堆栈”这种大容器来保存函数的“栈帧”小容器?看下面这个 C 语言的例子:

清单: test.c

作用: 演示栈帧

```
int add(int, int);
```

```
int main(){
```

```
    int m = 5;
```

```
    int n = 3;
```

```
    int z = add(m, n);
```

```
    return 0;
```

```
}
```

```
int add(int m, int n) {
```

```
    int z = m + n;
```

```
    return z;
```

```
}
```

这个例子很简单，`add()`函数用于求和。编译后，这段程序会生成对应的机器码。站在机器的角度看，要让 `main()`函数成功调用到 `add()`函数，需要解决下面 4 个主要问题：

- ◎ 当 CPU 从 `main()`函数跳转到 `add()`函数时，CPU 要能够拿到 `add()`函数的机器指令的位置，否则 CPU 无法执行 `add()`函数的机器指令。
- ◎ 当 `add()`函数执行完成之后，CPU 要能够重新拿到 `main()`函数的机器指令，以便跳转回 `main()`函数继续执行。
- ◎ 当 CPU 从 `main()`函数进入 `add()`函数之后，CPU 需要为 `add()`函数里的变量分配内存空间，以便在内存中存储 `add()`函数内部的局部变量的值。
- ◎ 当 CPU 从 `add()`函数返回 `main()`函数之前，由于 `add()`函数已经完成其使命，其内部的局部变量失去了意义，并且外部也压根儿访问不到这些局部变量，因此 CPU 需要将 `add()`函数的局部变量回收掉，以便让宝贵的内存空间能够重复被使用。

在考虑这 4 个问题，让我们忘记所谓的栈帧，忘记一切，我们就是最原始的 CPU 设计师。在那个时代，还没有所谓堆栈的说法问世呢！

作为一个 CPU，面前广阔无垠的内存空间可以任意驱驰，身后跟着 `ax`、`bx`、`cx` 等一众小弟，它们的名字叫作寄存器。CPU 的主要任务就是读取一条机器指令，然后执行，然后再接着读取下一条指令，再执行，再读取下一条指令再执行……，如此循环往复。上面第 1 个问题和第 2 个问题比较好解决，当加载程序时，只需将 `main()`函数和 `add()`函数的机器指令分别保存到内存的两个地方，并且 CPU 能够拿到这两个地方的内存地址，就可以了。当 CPU 执行完 `main()`函数的最后一条机器指令后，能够得到 `add()`函数的第一条机器指令，这样 CPU 自然而然就转去执行 `add()`函数的机器指令。事实上在编译阶段，编译器就将 `main()`函数和 `add()`函数的内存地址以标号这种相对位移的方式（专业的说法叫代码段）保存在了内存中，所以 CPU 一定能够得到这两个地址。现在关键就是第 3 和第 4 这两个问题了。

按道理说，CPU 可以将 `main()`函数和 `add()`函数所对应的栈帧（即堆栈空间）分配到内存中的任意两个位置，而随意分配的策略也有很多种，例如可以根据函数名进行散列而得到一个内存地址，然后从这个地址开始为函数内部的局部变量分配内存。但是这种方式即使从表面看都能发现很多问题，例如内存中的很多碎片空间都是有用的，可能存放了其他程序或其他函数的机器指令，可能存放了程序的全局变量或静态变量，也可能用作其他程序的堆栈空间，因此通过随机散列的方式获取内存地址将会使事情变得十分危险，不过如果真有人愿意尝试这种危险的方案，也一定能够设计出相对应的策略，假设函数堆栈真的按照这种方式去分配，最终一个程序的若干函数的堆栈空间布局就会像图 7.8 所示的这样。

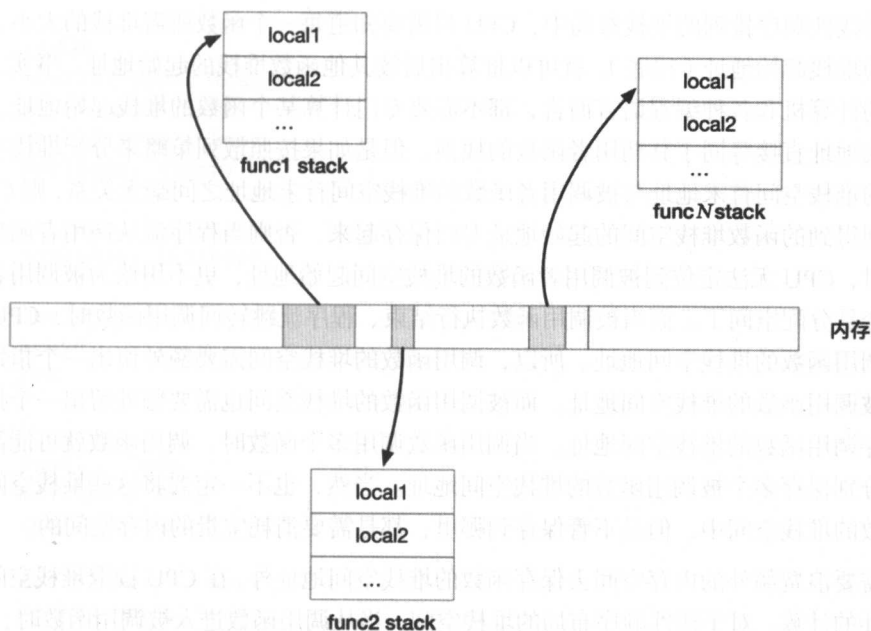


图 7.8 散列策略下的堆栈空间布局

可以看到，这种策略下的堆栈空间彼此之间是割裂的，毫无联系，因为是随机散列的嘛。这种空间分配策略最终也形成一个大容器，结构类似于 Java 语言中的 `hashmap`。不过空间是否割裂，这并没有任何关系，只要能实用就行。但是这种碎片化的分配策略与线性的分配策略相比，除了会造成 CPU 更多的计算（要进行散列计算）外，还会造成更多的存储空间浪费。这种浪费体现在 CPU 对每一个堆栈的首地址的记忆上。图 7.9 所示是一种线性的、顺序分配的堆栈布局图。

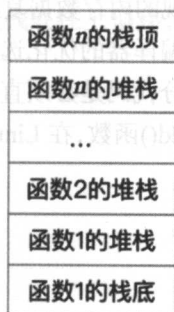


图 7.9 线性顺序分配的堆栈布局

在这种线性顺序排列的堆栈布局中，CPU 只需要知道每一个函数所需堆栈的大小，以及第一个函数的堆栈起始地址（栈底），就可以推算出后续其他函数堆栈的起始地址。事实上，对于现代成熟的计算机和各种编程语言而言，都不需要专门计算某个函数的堆栈起始地址，每一个函数的栈底地址直接等同于其调用者函数的栈顶。但是如果按照散列策略来分配堆栈空间，调用者函数的堆栈空间首末地址与被调用者函数的堆栈空间首末地址之间毫无关系，则 CPU 需要将每次散列得到的函数堆栈空间的起始地址专门保存起来，否则当程序流从调用者函数跳转到被调用者时，CPU 无法定位到被调用者函数的堆栈空间起始地址，更不用谈为被调用者函数内部的局部变量分配空间了。而当被调用函数执行结束、程序流跳转回调用函数时，CPU 需要重新定位到调用函数的堆栈空间地址。所以，调用函数的堆栈空间需要额外留出一个指针的宽度用于保存被调用函数的堆栈空间地址，而被调用函数的堆栈空间也需要额外留出一个指针的宽度用于保存调用函数的堆栈空间地址。当调用函数调用多个函数时，调用函数就可能需要留出若干空间分别保存多个被调用函数的堆栈空间地址。当然，也不一定要将这些堆栈空间的地址保存在函数的堆栈空间中，但是不管保存到哪里，都是需要消耗宝贵的内存空间的。

除了需要浪费额外的内存空间去保存函数的堆栈空间地址外，在 CPU 读取堆栈空间地址时也需要额外的计算。对于线性顺序布局的堆栈空间，当从调用函数进入被调用函数时，由于 SP 寄存器保存了调用函数的栈顶地址，因此 CPU 可以直接通过“mov %sp, %bp”这种纯粹寄存器数据传送的方式读取到被调用函数的栈底，在计算机内部，没有什么比纯寄存器之间的直接数据传送速度更快的了。而在随机散列策略中，由于被调用函数的堆栈空间地址被保存在内存中，因此无论如何，都避免不了 CPU 对内存的访问，而数据在内存和寄存器之间的传送速度往往要比数据直接在寄存器之间的传送效率低百倍以上。

更进一步，随机散列策略除了时间和空间上的低效，对堆栈的优化也被死死束缚。在 Java 虚拟机中，对堆栈有一项重要的优化策略是堆栈重叠（调用方法的操作数栈与被调用方法的局部变量表重叠），这种堆栈重叠的方式能够避免调用函数内部局部变量向被调用函数堆栈复制被调用函数的入参，从而能够减少相当可观的内存数据复制。这种现象不仅 Java 虚拟机中会有，其他编程语言中也会有，例如一些主流编译器的优化选项都支持这种策略。更有甚者，当在开发底层驱动器或者视频核心逻辑时，部分代码是必须直接用汇编写的，这时候就可以手动直接让堆栈重叠，例如上面的 main() 函数和 add() 函数，在 Linux 平台上编译后得到的汇编程序如下：

```
main:
    pushl    %ebp
    movl %esp, %ebp
    subl $32, %esp

    // 分配局部变量并初始化
    movl $5, 20(%esp)
    movl $3, 24(%esp)
```

```

//压栈
movl24(%esp), %eax
movl%eax, 4(%esp)
movl20(%esp), %eax
movl%eax, (%esp)

calladd
movl%eax, 28(%esp)

movl$0, %eax
leave
ret

add:
    pushl    %ebp
    movl%esp, %ebp
    subl$16, %esp

    movl12(%ebp), %eax
    movl8(%ebp), %edx
    addl%edx, %eax

    movl%eax, -4(%ebp)
    movl-4(%ebp), %eax
    leave
    ret

```

上面这段汇编脚本比较中规中矩,当 main()函数调用 add()函数并向 add()函数传递参数时,老老实实地将 add()函数的入参复制了一遍,由于 add()函数包含两个入参,因此一共使用了 4 条指令完成两个参数的压栈。

如果这段程序由人工编写,则可以进行比较激进的优化,省略复制参数的 4 条指令。在 add()函数中读取入参时,直接基于 add()堆栈栈底往上偏移,进入 main()函数的堆栈空间读取入参。激进优化后的汇编指令如下:

```

main:
    pushl    %ebp
    movl%esp, %ebp
    subl$16, %esp

    //分配局部变量并初始化
    movl$5, 4(%esp)
    movl$3, 8(%esp)

    calladd

```



```

movl%eax, 12(%esp)

movl$0, %eax
leave
ret

add:
    pushl    %ebp
    movl%esp, %ebp
    subl$16, %esp

    //从main()函数堆栈中直接读取局部变量
    movl12(%ebp), %eax
    movl16(%ebp), %edx

    addl%edx, %eax
    movl%eax, -4(%ebp)
    movl-4(%ebp), %eax
    leave
    ret

```

修改之后，main()函数少了 4 条参数复制的指令，同时 main()函数的堆栈空间只需要分配 16 字节，而原来是 32 字节。add()函数直接从 main()函数的堆栈中入参的原始内存位置处读取数据。这种优化方案所带来的性能提升还是相当可观的，这就是线性顺序内存分配所带来的好处，只要你想得到，就能做得到。而采用随机散列的方式分配函数堆栈，无法享受这些优化措施所带来的性能提升红利。

由于线性顺序分配栈帧（堆栈空间）有这么多好处，因此成为设计栈帧容器的上佳策略。而线性顺序存储元素成员的容器有好几种，最主要的两种容器分别是堆式容器（队列）和栈式容器，到了这里大家都知道了，堆与栈的区别就是前者是 FIFO（先进先出），后者是 FILO（先进后出）。由于函数调用过程是链式路径，越是最后调用的函数，其栈帧（堆栈空间）就越分配得晚；越是最后调用的函数，其栈帧空间也越早被释放，所以自然就使用栈式结构来作为栈帧的容器。这就是堆栈的由来。

到了这里，还有个问题没有解决，那就是操作系统一般都是从高位地址开始往下扩展堆栈空间。堆栈作为一个内存容器，并不一定非得从高位地址开始往下增长，也可以从低地址开始往高地址内存方向增长，有部分操作系统就是这么干的。而更重要的一点，对于一个应用程序，操作系统明显地会将程序的内存空间区分为堆和栈（当然还有代码区等，这些对于本主题并不是十分重要，略过不表），堆区往高地址方向增长，栈区往低地址方向增长，这又是何呢？换句话说，操作系统完全可以不区分所谓的堆区和栈区，大家都只是内存中的一部分空间而已，管它是啥结构，一股脑儿全分配在内存中，也不是不可行。软件程序中的所谓堆和栈的容器

就全部分配在堆内存中，也没见有啥问题。

事实上，这里需要区分操作系统和应用程序的内存结构。对于操作系统，其上面会运行若干应用程序，如果操作系统不将应用程序的堆内存和栈内存区分开来，那么内存空间就会相互“打架”，并且会破坏函数栈帧在空间上的线性连续性。还是拿上面的 `main()` 函数和 `add()` 函数举例，假设 `main()` 函数的堆栈从内存为 0 的地址处开始分配空间（这种情况事实上是不存在的，这里仅仅为了举例方便），`main()` 函数堆栈需要 32 字节的空间，因此当 `main()` 函数的堆栈空间分配完之后，下一个数据只能从第 32 内存单元开始分配空间。假设这时 `main()` 函数调用了 `alloc()` 函数向操作系统申请内存空间，那么操作系统就会从第 32 个内存单元开始分配空间（注意，这种假设的前提是操作系统没有将应用程序的内存区分为堆内存和栈内存），等分配完了 `alloc()` 函数所申请的内存空间后，`main()` 函数开始调用 `add()` 函数，此时 CPU 会为 `add()` 函数分配栈空间，而栈空间的起始位置就是刚才 `alloc()` 函数所申请的内存的末端位置。这样一来，`main()` 函数与 `add()` 函数的栈帧（堆栈空间）在内存空间上的线性顺序性就被破坏了，如此一来，调用函数与被调用函数的栈帧在空间上便处于割裂状态，这与上面所举的散列方案所产生的内存空间布局也没什么两样。所以，这就要求存储程序函数栈帧的容器——堆栈，在内存空间分布上必须是完全线性的，中间不能出现任何空间分隔。既然如此，操作系统就必须将应用程序的堆内存空间与栈内存空间完全隔离开来，将此作为操作系统治理内存的基本宗旨，而这也是程序函数堆栈这个容器级别的数据结构与软件程序中的堆栈容器的数据结构之间最大的不同点。

在软件程序中设计的所谓堆栈容器，栈中的元素也可以是容器，但是栈中往往仅仅保存对应元素的容器的指针，指针所指向的容器在堆中开辟空间，这便导致各个元素所指向的容器空间之间并不是连续的线性分布，例如，下面这个程序是用 C 语言所编写的堆栈容器，堆栈的元素是栈帧，栈帧也是一个容器：

清单：stack.c

作用：使用 C 语言演示堆栈布局

```
#include <stdio.h>
#include <stdlib.h>

//栈帧结构体，假设所有函数的栈帧都一样，都仅包含 3 个布局变量
typedef struct Frame
{
    struct Frame *pre; //指向上一个栈帧
    int local1;
    int local2;
    short local3;
} frame;

//堆栈
```

```
typedef struct Stack
{
    frame *base;//栈底
    frame *top;//栈顶
    int size;//堆栈大小
}stack;

void initStack(stack *stack);//初始化
void push(stack *mystack, frame *nextFrame);
void pop(stack *mystack);
void iterate(stack *mystack);

int main(int argc, char const *argv[])
{
    //定义一个堆栈
    stack *mystack = (stack*)malloc(sizeof(struct Stack));
    printf("sizeof(struct Frame) = %lu\n", sizeof(struct Frame));

    //初始化堆栈
    initStack(mystack);

    //压入第 1 个栈帧
    frame *myframe = (frame*)malloc(sizeof(struct Frame));
    printf("当前栈帧起始地址是 %p\n", myframe);
    myframe->local1 = 1;
    myframe->local2 = 2;
    myframe->local3 = 3;
    push(mystack, myframe);

    //压入第 2 个栈帧
    myframe = (frame*)malloc(sizeof(struct Frame));
    printf("当前栈帧起始地址是 %p\n", myframe);
    myframe->local1 = 5;
    myframe->local2 = 6;
    myframe->local3 = 7;
    push(mystack, myframe);

    //压入第 3 个栈帧
    myframe = (frame*)malloc(sizeof(struct Frame));
    printf("当前栈帧起始地址是 %p\n", myframe);
    myframe->local1 = 8;
    myframe->local2 = 9;
    myframe->local3 = 10;
    push(mystack, myframe);

    //遍历堆栈
    iterate(mystack);
}
```

```

//出栈
pop(mystack);
iterate(mystack);

pop(mystack);
iterate(mystack);

pop(mystack);
iterate(mystack);

pop(mystack);

return 0;
}

```

//初始化堆栈

```

void initStack(stack *stack)
{
    stack->base = NULL;
    stack->top = NULL;
    stack->size = 0;
}

```

//压栈

```

void push(stack *mystack, frame *nextFrame)
{
    //如果当前堆栈为空，则将栈顶和栈底同时指向第一个栈帧
    if(mystack->size == 0){
        mystack->base = nextFrame;
        mystack->top = nextFrame;
    }

    nextFrame->pre = mystack->top; //当前栈帧的上一个栈帧是栈顶
    mystack->top = nextFrame; //栈顶变成了当前栈帧
    mystack->size ++; //将堆栈数量加1
}

```

//出栈

```

void pop(stack *mystack)
{
    int size = mystack->size;
    if(size == 0)
    {
        printf("%s\n", "当前堆栈为空，不能出栈");
        return;
    }
}

```

```

    }

    frame *currFrame = mystack->top; //获取当前堆栈顶部栈帧
    mystack->top = mystack->top->pre; //将当前栈顶指向上一个栈帧

    //释放栈顶栈帧
    free(currFrame);
    mystack->size --;
    printf("成功弹出一个栈顶栈帧\n\n");
}

//遍历堆栈
void iterate(stack *mystack)
{
    if(mystack->size == 0){
        printf("%s\n", "当前堆栈为空，遍历终止");
        return;
    }

    int size = mystack->size;
    printf("*****当前堆栈共有 %d 个栈帧*****\n", size);
    frame *currFrame = mystack->top;
    while(size > 0)
    {
        printf("当前是第 %d 个栈帧\n", size);
        printf(" local1 = %d\n", currFrame->local1);
        printf(" local2 = %d\n", currFrame->local2);
        printf(" local3 = %d\n", currFrame->local3);

        currFrame = currFrame->pre;
        size --;
    }
}

```

本示例程序中定义了两个结构体，分别是 `stack` 与 `frame`，`stack` 是堆栈容器，里面包含栈底、栈顶和堆栈大小 3 个成员项。`frame` 则用于构建栈帧。运行 `main()` 函数，会打印如下内容：

```

sizeof(struct Frame) = 24
当前栈帧起始地址是 0x7fa3d3c02000
当前栈帧起始地址是 0x7fa3d3c03290
当前栈帧起始地址是 0x7fa3d3c032b0
*****当前堆栈共有 3 个栈帧*****
当前是第 3 个栈帧
    local1 = 8
    local2 = 9
    local3 = 10
当前是第 2 个栈帧

```

```

local1 = 5
local2 = 6
local3 = 7

```

当前是第 1 个栈帧

```

local1 = 1
local2 = 2
local3 = 3

```

成功弹出一个栈顶栈帧

*****当前堆栈共有 2 个栈帧*****

当前是第 2 个栈帧

```

local1 = 5
local2 = 6
local3 = 7

```

当前是第 1 个栈帧

```

local1 = 1
local2 = 2
local3 = 3

```

成功弹出一个栈顶栈帧

*****当前堆栈共有 1 个栈帧*****

当前是第 1 个栈帧

```

local1 = 1
local2 = 2
local3 = 3

```

成功弹出一个栈顶栈帧

当前堆栈为空，遍历终止

当前堆栈为空，不能出栈

注意看程序所打印出来的 3 个栈帧的内存地址，这 3 个地址之间没有任何联系，这是因为这 3 个栈帧完全是操作系统随机分配的内存空间。所以说，软件程序里所定义的栈结构，与操作系统层面的程序函数堆栈结构之间还是有差别的。本例动态创建的 3 个栈帧，内存布局完全是无序和随机的，它们的内部空间结构如图 7.10 所示。

而程序函数堆栈中的各个栈帧之间是完全紧密顺序排列的，栈帧之间不可能出现别的数据。

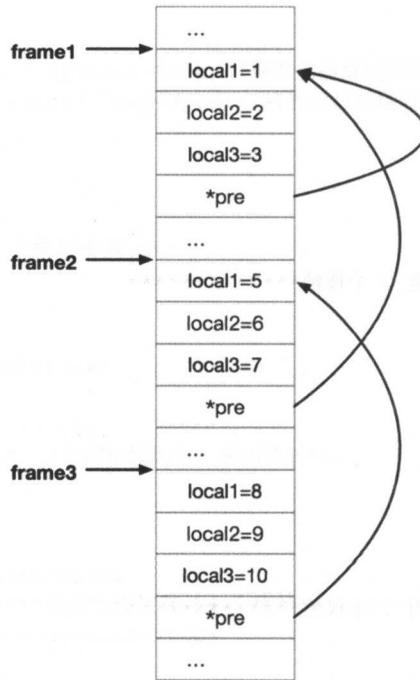


图 7.10 C 语言所模拟的随机分配的堆栈空间布局

当然，使用软件其实也能模拟出程序函数堆栈的空间分布，将上面的程序进行改造，使之分配的栈帧彼此首尾相连，改造后的程序如下：

清单：stack.c

作用：使用 C 语言演示堆栈布局

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
//堆栈初始大小
```

```
#define DEFAULT_SIZE 2
```

```
//堆栈扩容因子
```

```
#define EXPAND_FACTORY 1.75
```

```
//栈帧结构体，假设所有函数的栈帧都一样，都仅包含 3 个局部变量
```

```
typedef struct Frame
```

```
{
```

```
    struct Frame *pre; //指向上一个栈帧
```

```
    int local1;
```

```
    int local2;
```

```
    short local3;
```



```

}frame;

//堆栈
typedef struct Stack
{
    frame *base;//栈底
    frame *top;//栈顶
    int size;//堆栈真实大小
    int maxSize;//堆栈最大大小
}stack;

void initStack(stack *stack);//初始化
void push(stack *mystack, frame *nextFrame);
void pop(stack *mystack);
void expand(stack *mystack);//扩容
void iterate(stack *mystack);

int main(int argc, char const *argv[])
{
    //定义一个堆栈
    stack *mystack = (stack*)malloc(sizeof(struct Stack));

    //初始化堆栈
    initStack(mystack);

    //压入第1个栈帧
    frame myframe;
    myframe.local1 = 1;
    myframe.local2 = 2;
    myframe.local3 = 3;
    push(mystack, &myframe);

    //压入第2个栈帧
    myframe.local1 = 5;
    myframe.local2 = 6;
    myframe.local3 = 7;
    push(mystack, &myframe);

    //压入第3个栈帧
    myframe.local1 = 8;
    myframe.local2 = 9;
    myframe.local3 = 10;
    push(mystack, &myframe);

    //压入第4个栈帧
    myframe.local1 = 11;
    myframe.local2 = 12;

```

```
myframe.local3 = 13;
push(mystack, &myframe);

//遍历堆栈
iterate(mystack);

//出栈
pop(mystack);
iterate(mystack);

pop(mystack);
iterate(mystack);

pop(mystack);
iterate(mystack);

pop(mystack);

free((void *)mystack->base);

return 0;
}

//初始化堆栈
void initStack(stack *stack)
{
    stack->size = 0;
    stack->maxSize = DEFAULT_SIZE;

    //初始化堆栈空间
    long address = (long)malloc(sizeof(struct Frame) * DEFAULT_SIZE);
    stack->base = (frame*)address;
    stack->top = (frame*)address;
}

//压栈
void push(stack *mystack, frame *nextFrame)
{
    //扩容
    expand(mystack);

    frame *newFrame;
    frame *top = mystack->top;

    //如果当前堆栈为空,则将栈顶和栈底同时指向第一个栈帧
    if(mystack->size == 0){
        newFrame = top;
    } else{
```

```

    newFrame = ++ top;
}

//frame *newFrame = (frame*)(++ top); //将栈顶指针往前移动一个栈帧的距离
printf("压入第 %d 个栈帧, 当前栈帧起始地址是 %p\n\n", mystack->size + 1,
newFrame);
newFrame->local1 = nextFrame->local1;
newFrame->local2 = nextFrame->local2;
newFrame->local3 = nextFrame->local3;

newFrame->pre = mystack->top; //当前栈帧的上一个栈帧是栈顶
mystack->top = newFrame; //栈顶变成了当前栈帧

mystack->size ++; //将堆栈数量加1
}

//出栈
void pop(stack *mystack)
{
    int size = mystack->size;
    if(size == 0)
    {
        printf("%s\n", "当前堆栈为空, 不能出栈");
        return;
    }

    frame *currFrame = mystack->top; //获取当前堆栈顶部栈帧
    mystack->top = mystack->top->pre; //将当前栈顶指向上一个栈帧

    //释放栈顶栈帧
    mystack->size --;
    printf("成功弹出一个栈顶栈帧\n\n");
}

//扩容
void expand(stack *mystack)
{
    if(mystack->size == mystack->maxSize){
        //扩容
        int maxSize = mystack->size * EXPAND_FACTORY; //计算扩容后的大小
        long address = (long)malloc(sizeof(struct Frame) * maxSize);
        memcpy((void *)address, mystack->base, mystack->size * sizeof(struct
Frame));

        //重定向栈顶和栈底
        mystack->base = (frame*)address;
        mystack->top = (frame*)(address + (mystack->size - 1) * sizeof(struct

```

```

Frame));

mystack->maxSize = maxSize;

//重新关联当前栈帧与上一个栈帧之间的关系
for(int i = 1; i < mystack->size; i++)
{
    frame *currFrame = (frame*)address;
    frame *tmp = (frame*)address;
    frame *nextFrame = ++tmp; //获取下一个栈帧
    nextFrame->pre = currFrame;

    address = (long)nextFrame;
}

printf(">>>>>扩容成功。申请的堆栈地址是 %p, 扩容后的堆栈栈顶地址是 %p\n",
mystack->base, mystack->top);
}

//遍历堆栈
void iterate(stack *mystack)
{
    if(mystack->size == 0){
        printf("%s\n", "当前堆栈为空, 遍历终止");
        return;
    }

    int size = mystack->size;
    printf("*****当前堆栈共有 %d 个栈帧*****\n", size);
    frame *currFrame = mystack->top;
    while(size > 0)
    {
        printf("当前是第 %d 个栈帧, 其地址是 %p\n", size, currFrame);
        printf("  local1 = %d\n", currFrame->local1);
        printf("  local2 = %d\n", currFrame->local2);
        printf("  local3 = %d\n", currFrame->local3);

        currFrame = currFrame->pre;
        size --;
    }
}

```

改造后的程序, 栈帧内存空间分配算法有所改变, 变成堆栈这个大容器统一分配一整块连续的内存, 然后在这块内存空间里按顺序存放栈帧。当堆栈容量不够时, 通过扩容重新分配一整块连续的内存空间, 然后再重新按顺序进行排列。

现在运行 main() 函数, 会打印出如下内容:

压入第 1 个栈帧, 当前栈帧起始地址是 0x7f96e1c03290

压入第 2 个栈帧, 当前栈帧起始地址是 0x7f96e1c032a8

>>>>扩容成功。申请的堆栈地址是 0x7f96e1c032c0, 扩容后的堆栈栈顶地址是 0x7f96e1c032d8
压入第 3 个栈帧, 当前栈帧起始地址是 0x7f96e1c032f0

>>>>扩容成功。申请的堆栈地址是 0x7f96e1c03310, 扩容后的堆栈栈顶地址是 0x7f96e1c03340
压入第 4 个栈帧, 当前栈帧起始地址是 0x7f96e1c03358

*****当前堆栈共有 4 个栈帧*****

当前是第 4 个栈帧, 其地址是 0x7f96e1c03358

local1 = 11

local2 = 12

local3 = 13

当前是第 3 个栈帧, 其地址是 0x7f96e1c03340

local1 = 8

local2 = 9

local3 = 10

当前是第 2 个栈帧, 其地址是 0x7f96e1c03328

local1 = 5

local2 = 6

local3 = 7

当前是第 1 个栈帧, 其地址是 0x7f96e1c03310

local1 = 1

local2 = 2

local3 = 3

成功弹出一个栈顶栈帧

*****当前堆栈共有 3 个栈帧*****

当前是第 3 个栈帧, 其地址是 0x7f96e1c03340

local1 = 8

local2 = 9

local3 = 10

当前是第 2 个栈帧, 其地址是 0x7f96e1c03328

local1 = 5

local2 = 6

local3 = 7

当前是第 1 个栈帧, 其地址是 0x7f96e1c03310

local1 = 1

local2 = 2

local3 = 3

成功弹出一个栈顶栈帧

*****当前堆栈共有 2 个栈帧*****

当前是第 2 个栈帧, 其地址是 0x7f96e1c03328

local1 = 5

local2 = 6

local3 = 7

当前是第 1 个栈帧，其地址是 0x7f96e1c03310

local1 = 1

local2 = 2

local3 = 3

成功弹出一个栈顶栈帧

*****当前堆栈共有 1 个栈帧*****

当前是第 1 个栈帧，其地址是 0x7f96e1c03310

local1 = 1

local2 = 2

local3 = 3

成功弹出一个栈顶栈帧

注意看程序所打印出来的 4 个栈帧的内存地址，现在这 4 个地址之间是有联系的，第 4 个栈帧的内存地址与第 3 个栈帧内存地址之差是 0x18，正好是 struct Frame 这个结构体的大小。同样，第 3 个栈帧内存地址与第 2 个栈帧的内存地址之差也是 0x18，第 2 个栈帧内存地址与第 1 个栈帧的内存地址之差也是 0x18，由此可见栈帧之间确实是线性按顺序分配空间的。这种堆栈的空间布局基本与程序函数调用的堆栈空间布局是一样的，布局结构如图 7.11 所示。

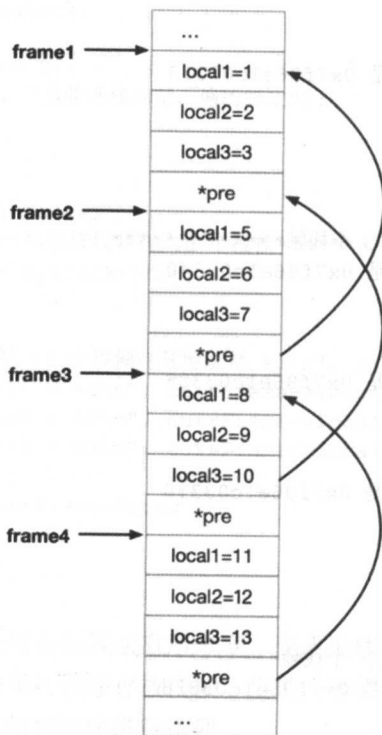


图 7.11 C 语言所模拟的线性顺序分配的堆栈空间布局

综上所述,程序函数调用的堆栈的成员元素是栈帧,“堆栈”作为一个容器,其内部存储的是“栈帧”这种元素。而栈帧又是一个小容器,存储函数内部的局部变量。函数调用堆栈与软件程序里所设计的堆栈结构的最大不同之处在于,函数调用堆栈的成员元素(栈帧)之间在空间上是连续分布的,而软件程序里所设计的堆栈结构,其成员元素往往并不相连。

7.3.2 硬件对堆栈的支持

为了支持应用程序函数栈帧所形成的堆栈结构,系统设计的老前辈们还从硬件上予以支持,专门设计了两个硬件寄存器——SP 和 BP,用于存储当前函数栈帧的栈底和栈顶。

在上面使用 C 语言模拟程序堆栈和栈帧的示例中,在结构体 `stack` 中专门定义了两个指针 `base` 和 `top`,分别指向堆栈的底部栈帧和顶部栈帧,注意这里不是栈顶指针和栈底指针哟!在操作系统层面,栈底和栈顶指针是针对栈帧而言的,例如,该示例中有 4 个栈帧 `frame`,如果是真实的函数栈帧,则每一个栈帧都会有一个栈顶和栈底指针,用于标识函数堆栈的起始地址和末端地址。而使用软件实现的栈帧,由于其数据结构是固定大小的,因此通过指针持有其起始地址,自然就知道其末端地址,何况使用软件实现堆栈和栈帧时,也不需要知道末端地址,直接通过强类型转换就能读取到栈帧各个变量的值,所以大部分使用软件实现的所谓堆栈,其栈底指针和栈顶指针更多的是用来指向整个堆栈容器的底部栈帧和顶部栈帧。如图 7.12 所示。

对于软件实现的堆栈,只需要获取指向堆栈顶部栈帧的指针,就能实现压栈和出栈的功能。而程序函数调用时为函数所开辟的堆栈空间,由于不同的函数其内部局部变量不同,因此栈帧空间结构不同,大小不同,因此必须要分别保存函数栈帧的栈顶地址与栈底地址。如果该示例中的 4 个栈帧是真实的函数调用堆栈,则 CPU 会分别使用 SP 和 BP 这两个寄存器存储当前位于堆栈顶部的栈帧的栈顶和栈底,结构示意图如图 7.13 所示。

其实如果没有 SP 与 BP 这两个硬件寄存器,系统也能完成对堆栈顶部函数栈帧的栈顶与栈底标记,但是那样势必要将栈顶与栈底地址保存到内存,而内存访问的性能与寄存器的访问性能完全不在一个档次。所以函数栈帧的空间分配其实用纯软件也能实现,只不过借助于两个特定的寄存器硬件,使得这一效率更高而已。

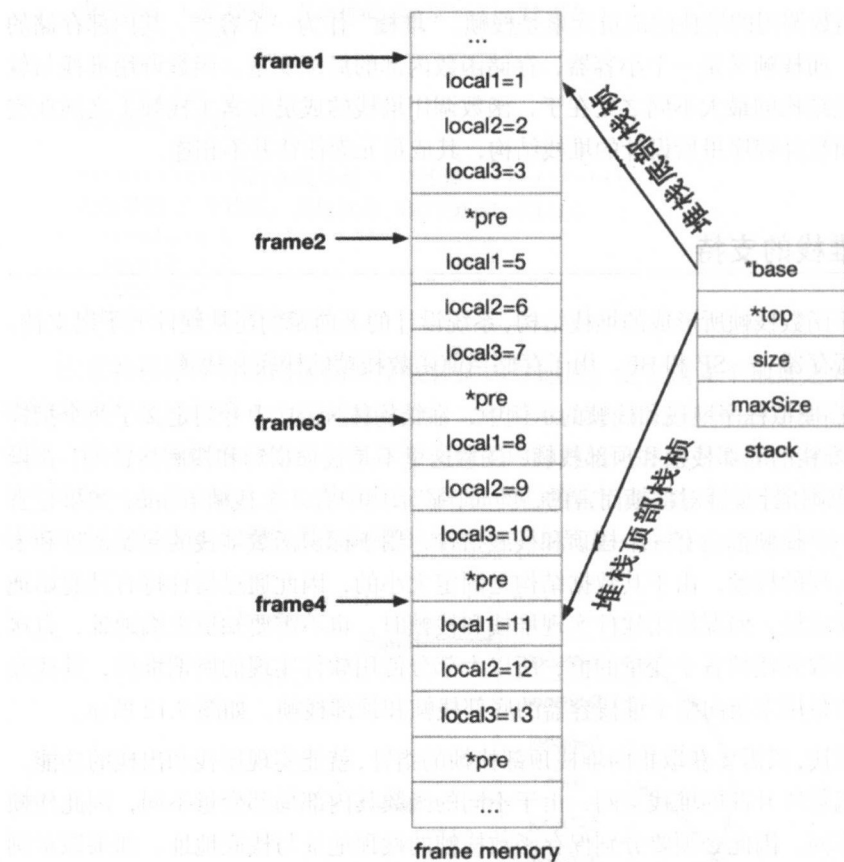


图 7.12 软件实现的堆栈，堆栈底部和顶部指针

当 CPU 完成一个函数的执行之后，程序流会跳转到当前函数的调用方，同时，CPU 需要回收掉当前函数的栈帧空间，并使 SP 和 BP 这两个寄存器重新指向调用者函数的栈顶和栈底。CPU 回收函数栈帧空间很简单，只需要将 SP 往 BP 的方向移动一定距离即可，但是 CPU 如何将 SP 与 BP 重定向至调用者函数的栈顶和栈底呢？其实也很简单，由于真实的程序函数调用堆栈里的各个栈帧都是首尾相连的，被调用函数的栈底就是调用函数的栈顶，因此当 CPU 完成函数调用后，只需将被调用函数的 SP 指向被调用函数的 BP，由于被调用函数的 BP 恰好就是调用函数的 SP，因此这就相当于 CP 将 SP 重新指向到了调用者函数的栈顶。所以剩下的事就是 CPU 如何将 BP 也恢复到调用者函数的栈底。

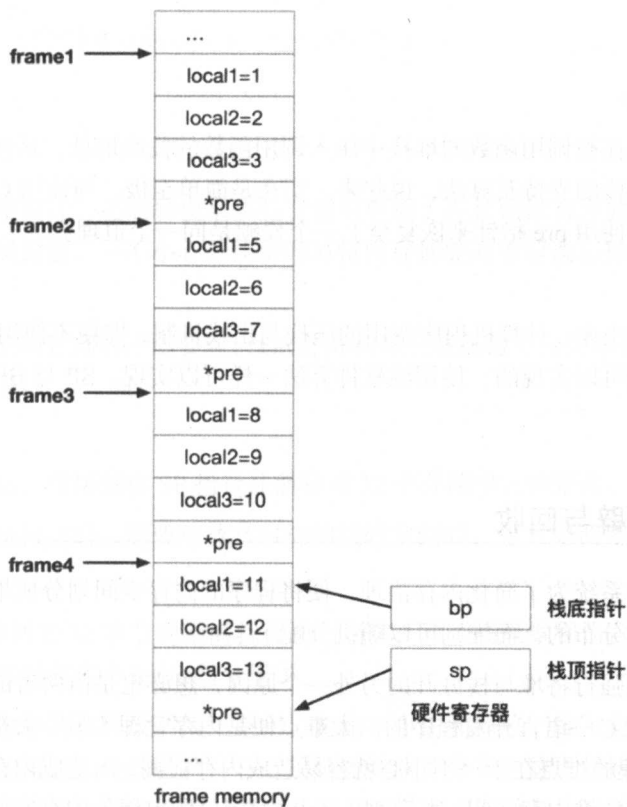


图 7.13 真实的硬件实现的栈底和栈顶指针

在上面使用 C 语言模拟函数调用堆栈和栈帧的示例中，为了实现出栈后能够定位到上一个栈帧，在每一个栈帧中都存储了一个 pre 指针，每次压入一个新的栈帧时，新的栈帧的 pre 指针就指向原本位于堆栈顶部的栈帧的起始地址，所以，当将位于堆栈顶部的栈帧出栈时，只需通过读取 pre 指针的值，就能定位到上一个栈帧，从而将堆栈的栈顶指针 top 重定向至新的堆栈顶部的栈帧。其实，在本示例中（特指上述改造后的示例程序，在改造后的示例程序中，各个栈帧是线性顺序分布的，首尾相连），由于所模拟的各个栈帧的数据结构相同，因此各个栈帧所占内存空间大小也相同，所以完全可以不依赖 pre 指针就可以直接计算出相邻的栈帧位置。

但是在机器的函数调用堆栈层面，由于不同的函数定义完全不同，因此函数的栈帧结构也完全不同，大小更不可能相同，所以当进行出栈时，想要将 BP 恢复至调用者函数的栈底，如果当前函数没有保存调用者函数的栈底地址，是根本无法恢复回去的。所以，在真实的函数调用中，物理机器必定会将调用函数的栈底地址压入到被调用函数的堆栈之中。因而，使用 C 语言及其他很多语言编写的程序被编译后，所生成的每一个函数的机器指令必定以下面这两条指

令作为“起式”：

```
push %bp
move %sp, %bp
```

第一条指令就是往被调用函数的堆栈中压入调用函数的栈底地址，这便是物理机器对程序调用堆栈的压栈与出栈的支持及算法，说起来，实在是简单至极，与使用 C 语言所模拟的堆栈和栈帧的示例程序中使用 `pre` 指针来恢复至上一个栈帧是同一个道理。

大道至简啊！

由此更加可以看出来，计算机程序调用的压栈与出栈机制，即使不使用 BP 和 SP 这两个硬件寄存器，也是完全可以实现的，使用纯软件算法一样可以实现。SP 与 BP 的加入，纯粹是为了提升效率。

7.3.3 栈帧开辟与回收

前面讲过，操作系统为了简化内存治理，便将程序的内存空间划分成堆和栈，并且栈必须在内存空间上是连续分布的，而堆则可以随机分配。

其实，操作系统强行将堆与栈分开的另外一个原因，想必也是由两者的空间分布的特点不同所引起的。使用 C/C++ 语言开发程序的一大难点便是内存管理（另一大难点自然就是多线程与并发了），内存管理的难点在于一不小心就容易造成内存泄漏，而造成内存泄漏的关键原因就是忘记释放已经申请的堆内存空间。对于使用 `malloc()` 接口所申请的内存空间，如果不使用 `free()` 去登记释放，除非程序运行结束，否则这部分申请的内存是不会被自动回收的。如果 `malloc()` 接口在一个多进程/多线程环境中被调用，那么要不了多久，操作系统内存空间便会耗尽而使系统崩溃。

下面这个程序就会造成内存泄漏：

清单：malloc_test.c

作用：演示内存泄漏

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char const *argv[])
{
    long k;
    while(1)
    {
        int *address = (int*)malloc(sizeof(int)*10000000);
```

```

    k++;
    printf("申请内存成功, k= %lu\n", k);
}

return 0;
}

```

对于简单的程序设计，还能看出来哪些变量只申请了内存而没有释放，而对于一个大型的程序而言，经过层层封装，一不小心，便很容易将内存释放的事情遗忘掉，所以开发 C/C++ 程序真是陷阱重重。

相比于堆内存空间管理的严酷要求，栈的开辟和释放就显得十分简单。想开辟一个栈空间，只需要调用如下指令：

```
sub $32, %sp
```

调用这条指令后，直接将 SP 指针往前移动 32 个存储单元的距离，相当简单。

而当一个函数执行完后，需要释放其对应的栈帧空间时，也只需调用下面的指令：

```
add $32, %sp
```

将 SP 指针再移回去 32 字节的距离就行了。当然，对于高级编程语言，这条指令基本是看不到的，因为直接被封装在了 leave 指令里了。

总之，应用程序申请和释放“栈”空间，相比于申请和释放“堆”空间，要简单得多。之所以如此简单，主要得益于栈的结构——必须是连续线性分布。这也是操作系统实现内存治理时要求将堆与栈空间分开治理的原因，试想一下，如果栈与堆没有分开治理而是合在了一起，那么当应用程序申请分配栈空间时势必也要使用类似于 malloc() 这样的接口，而当函数调用完成，操作系统需要回收该函数所占用的栈帧空间时，势必也需要调用类似于 free() 这样的接口，如此一来，在函数栈帧压栈与出栈的过程中，还要涉及系统调用，那效率是相当低啊！

所以，不管是为了能够高效实现栈帧的压栈与出栈而将堆与栈空间相分离，还是纯粹为了堆与栈分开治理而发现了分离后能够享受到高效的压栈与出栈算法所带来的性能提升，总之，将应用程序的堆与栈隔离开来，并且规定栈帧在堆栈空间内必须线性连续分布，而且还借助于 SP 与 BP 这两个硬件寄存器来实现高效的压栈与出栈指令，都是历史上那些伟大的软件设计师经过千锤百炼后所得出的最优方案，是精华中的精华。

7.3.4 堆栈大小与多线程

现代操作系统都支持多进程与多线程机制，允许同一个应用程序同时运行若干进程或者线程，作为虚拟操作系统的 JVM 自然也是支持多进程与多线程应用的。

沿着上文对堆栈与栈帧认识的思路，可以继续往下推断在多线程与多线程机制下的栈空间分配机制，对于这些机制，不用看任何理论，也不用看操作系统原理相关的书，完全借助于上文所得到的结论便可分析。前面讲过，操作系统会将一个应用程序的内存空间划分成堆和栈这两大部分，函数的栈帧只能在栈空间内分配。但是在多线程/多线程环境下，就会有一个关键的问题：在多线程或多线程的环境下，操作系统会为一个进程/线程单独划分一个栈空间，还是会让一个应用程序的所有进程/线程共同使用同一个栈空间呢？说白了就是，在多线程环境下，操作系统是否会进一步将应用程序的栈空间划分成更多的子块？

我们可以先从一个假设出发去论证。假设操作系统不将应用程序的栈空间按照线程的粒度进行细分，而是让全部线程共同使用同一个堆栈空间，那么随着一个应用程序的多个线程交替执行，该应用程序的堆栈空间布局就可能会变成如图 7.14 所示的样子。

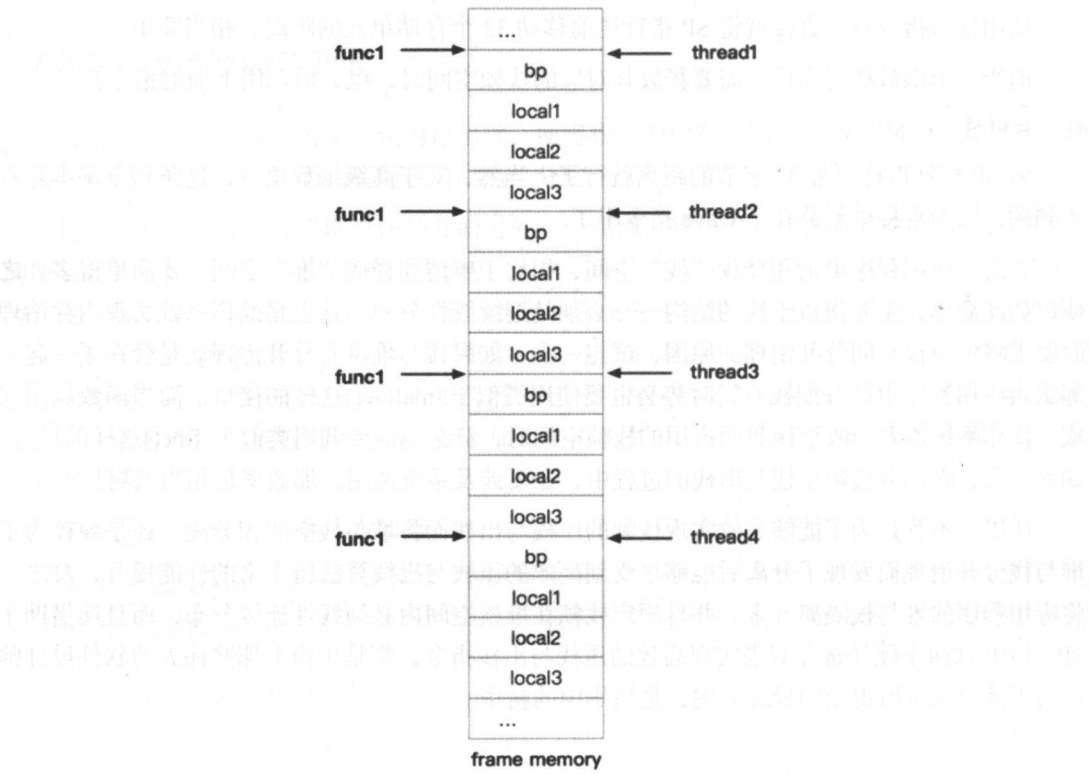


图 7.14 多线程共用同一个堆栈空间时的一种可能的栈帧布局

注：为了简化问题分析，本图在栈帧之间省略了除 bp 以外的其他堆栈所必需的数据。

假设该应用程序同时运行了 4 个线程，并且恰好这 4 个线程依次执行了 `func1()` 这个函数，那么此时的堆栈内的栈帧空间布局就会如图 7.14 所示。

当出现这种内存布局时，就违反了操作系统关于堆栈空间布局的最核心的原则：栈帧必须是连续分布的。例如对于线程 1，其 `func1` 的下一个栈帧应该是 `func2`（假设 `func1` 调用了 `func2`），但是现在却变成了线程 2 的 `func1()` 函数的栈帧。

这样不连续的内存布局无论是对新函数的栈帧分配还是函数执行结束时的栈帧回收，都会造成相当大的问题，程序的堆栈空间变得杂乱无章，各个线程的堆栈空间相互覆盖重写，世界一片大乱。

所以，为了支持多线程应用，操作系统必须为应用程序的每一个线程专门划分一块连续的区域，各个线程的函数调用堆栈就在各自的内存区域内按照单一方向进行压栈或出栈。所以，真实的多线程应用程序的内存空间布局会是如图 7.15 所示的情形。

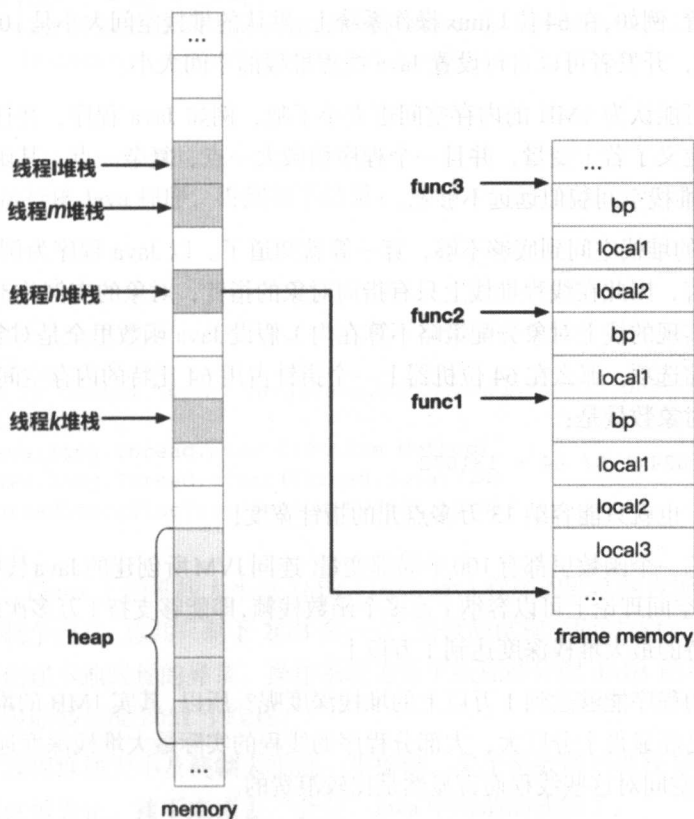


图 7.15 真实的多线程应用程序的堆栈分布图

与一个应用程序能够运行多线程同理，一个操作系统上会运行若干应用程序，为了让每一个应用程序的堆栈空间能够独自完整地遵循线性顺序分布的原则（中间不允许有割裂），其实操作系统在加载一个应用程序时，就会同时划分一片内存区域给这个应用程序作为其堆栈之用，这块内存区域与其他应用程序的不会重叠。

既然操作系统为每一个应用程序划分了一块独立的、连续的内存区域，同时还为一个应用程序的每一个线程/进程也划分了一块独立的、连续的内存区域，那么问题又来了，内存空间是有限的，而应用程序和线程却可以有很多，例如一个生产环境中的 JVM 服务器就会有两三千个线程，何况这仅仅是一个 JVM 进程而已，操作系统上还要运行其他若干应用程序，而每个应用程序中又会有若干线程。所以，大量的线程与有限的内存容量之间形成了一对尖锐的矛盾，换言之，线程堆栈的空间不是任意大的，一定是会有约束和限制的，否则如果随便一个线程堆栈空间都有一两百兆，那么一个 2GB 的内存空间也只能同时运行几十个线程，很显然这是肯定无法满足现代操作系统和大型应用程序的需求的。所以，几乎所有主流的操作系统都会有默认堆栈空间大小的设置。例如，在 64 位 Linux 操作系统上，默认的堆栈空间大小是 1024 KB，即 1MB。而对于 JVM 而言，开发者可以自行设置 Java 线程堆栈的空间大小。

有的小伙伴可能认为 1MB 的内存空间也太小了吧，例如 Java 程序，往往一个稍微复杂点的函数，里面就定义了若干变量，并且一个程序稍微大一点、复杂一点，其函数调用堆栈都是很深的，1MB 的堆栈空间貌似远远不够吧。

别急，1MB 的堆栈空间到底够不够，算一算就知道了。以 Java 程序为例，由于 Java 是面向对象的编程语言，因此在线程堆栈上只有指向对象的指针，对象的实例并不在堆栈上（JVM 为了优化而内部实现的栈上对象分配策略不算在内）。假设 Java 函数里全是对象，并且假设 JVM 没有开启指针压缩选项，那么在 64 位机器上一个指针占用 64 比特的内存空间，1MB 内存空间可以容纳的指针对象数量是：

$$1 * 1024 * 1024 * 8 / 64 = 131072$$

呵呵，不多，也就只能容纳 13 万多点儿的指针宽度！

假设程序的每一个函数里都有 100 个局部变量（连同 JVM 所创建的 Java 栈帧持有的数据），那么 1MB 的堆栈空间理论上可以容纳 1 万多个函数栈帧，即能够支持 1 万多次的函数顺序调用。专业一点就是支持的最大堆栈深度达到 1 万以上。

什么样复杂的程序能够达到 1 万以上的堆栈深度呢？所以，其实 1MB 的堆栈空间对于绝大多数函数而言，已经显得十分巨大，大部分程序的线程的实际最大堆栈深度远远达不到 1 万，所以 1MB 的堆栈空间对这些线程而言显然是比较浪费的。

对于 Java 应用程序而言，可以使用下面的程序测试堆栈大小设置：

清单: ThreadStackSizeTest.java

作用: 测试 Java 程序堆栈大小

```
import java.util.concurrent.CountDownLatch;
```

```
public class ThreadStackSizeTest extends Thread{
    CountDownLatch cdl = new CountDownLatch(1);
```

```
    public void run(){
        try {
            cdl.await();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

```
    public static void main(String[] args){
        for(int i = 0; ; i++){
            new Thread(new ThreadStackSizeTest()).start();
            System.out.println("count=" + i);
        }
    }
}
```

以默认设置运行该 Java 程序, 得到如下结果:

```
count=0
.....
count=2019
count=2020
count=2021
Exception in thread "main" java.lang.OutOfMemoryError: unable to create new
native thread
    at java.lang.Thread.start0(Native Method)
    at java.lang.Thread.start(Thread.java:714)
    at ThreadStackSizeTest.main(Student.java:20)
    at .....
```

默认设置下, Linux 上的堆栈大小是 1MB, 因此 Java 程序的默认线程堆栈大小也是 1MB。查看操作系统的剩余内存, 总共只剩下 2GB 多一点, 而该测试程序同时运行了 2021 个线程后, 终于抛出不能再创建本地线程的异常。操作系统为每个线程都分配 1MB 的堆栈空间, 2GB 内存空间正好差不多能够分配 2021 个线程。

下文继续对线程堆栈大小及栈帧大小进一步探讨, 关于函数调用堆栈与栈帧的普及知识讲了很多了, 先到这里为止。接下来该去“会会”Java 中所谓的栈帧了。

7.4 JVM 的栈帧

7.4.1 JVM 栈帧与大小确定

Java 虚拟机是建立在物理操作系统之上的虚拟系统，其栈帧与物理操作系统上的栈帧有所不同。物理操作系统直接基于 CPU 硬件执行指令，而 Java 虚拟机则只能基于栈式指令集运行，两者不同的运行机制决定了栈帧结构的不同。但是，Java 虚拟机本身其实并没有真正的执行能力，说到底最终还是不得不调用 CPU 硬件指令去完成程序逻辑，所以 Java 函数的栈帧与物理机器的栈帧之间又存在莫大的内在联系，这种联系体现在两方面。

1. 堆栈的设计

Java 函数的堆栈设计直接借用物理操作系统的堆栈管理思想，并无创新。Java 虚拟机也将一个 Java 应用程序的内存空间划分成堆内存与栈内存，分别治理。

另一方面，Java 函数的调用链路既然也使用“堆栈”这种容器级别的数据结构，本身就决定了 Java 应用程序的堆与栈必须要分别划分，这一点在上文进行过逻辑推导。

2. 栈帧的设计

堆栈作为栈帧的容器，其设计思路一旦确定，栈帧的设计也必然随之被确定，逃不出“压栈”与“出栈”这种“三界轮回”的铁的法则，否则 Java 函数调用链路的容器就不是“堆栈”这种数据结构了。

既然 Java 函数的内部数据使用“栈帧”这种容器进行存储，则必然导致 Java 函数的栈帧也要存储 Java 函数内的局部变量。

同时，上文也推导过，当一个操作系统决定使用“堆栈”这种容器对“栈帧”这种子容器进行线性顺序存储的策略时，由于不同函数其内部所包含的局部变量类型和数量都不同，因此栈帧大小也不同，为了支持当一个函数被执行完成后，能够让 CPU 重新定位到该函数的调用者函数的堆栈中去，一种最简单的算法就是在被调用函数的堆栈中保存调用函数的栈底地址，既然最简单的算法，Java 虚拟机当然也得借用这种算法。所以，Java 栈帧除了需要保存 Java 方法内的局部变量外，还需要保存一些支持堆栈开辟与回收的上下文数据。Java 栈帧里保存 Java 方法内部局部变量的区域叫作“局部变量表”，而 Java 栈帧里保存上下文数据的区域叫作“JVM 帧数据”。

所以，现在知道 Java 栈帧至少由局部变量表和 JVM 帧数据所组成。

众所周知, Java 的指令集是面向栈的, 是栈式指令集。与栈式指令集相对的是寄存器指令集。其中寄存器指令集直接被 CPU 硬件所支持。对于基于寄存器指令集所设计的软件程序, 其逻辑处理皆与寄存器紧密相关, 一点都脱不开干系, 例如下面的示例程序:

清单: test.s

作用: 演示寄存器式指令集

```
add:
    pushl    %ebp
    movl%esp, %ebp
    subl$16, %esp

    //读取第 1 个入参
    movl12(%ebp), %eax

    //读取第 2 个入参
    movl8(%ebp), %edx

    //对第 1 和第 2 这两个入参进行累加
    addl%edx, %eax

    movl%eax, -4(%ebp)
    movl-4(%ebp), %eax
    leave
    ret
```

本程序使用汇编语言定义了一个 add(int, int)函数, 在函数里面无论是读取入参, 还是对两个整数进行求和, 皆需要通过寄存器方能完成。

但是 Java 语言不是这种机制, 对于下面这段 Java 程序:

清单: JavaStack.java

作用: 演示 Java 的栈式指令集

```
class JavaStack{
    public static int add(int x, int y){
        int z = x + y;
        return z;
    }
}
```

编译后得到的字节码指令如下:

```
public static int add(int, int);
```

Code:

```
Stack=2, Locals=3, Args_size=2
```

```
0:   iload_0
```

```
1:   iload_1
```

```

2:    iadd
3:    istore_2
4:    iload_2
5:    ireturn
LineNumberTable:
  line 14: 0
  line 15: 4

```

诸君可以看到，Java 语言被编译后生成的指令，没有一条是与硬件寄存器相关的，反而大部分都是围绕栈的指令，例如，`iload` 将数据压栈，`istore` 将栈顶数据弹出，`iadd` 则对栈顶的两个数据进行累加，等等。

直接基于寄存器的指令之所以能够操纵寄存器，那是因为每个寄存器都能用来存储数据，只不过仅能存储一个数据。寄存器指令对寄存器的操作其实就是对相应寄存器中的数据进行的操作。同理，栈式指令集若想对栈中的数据进行操作，前提就是得有一部分内存能够被用作栈空间，如此，栈式指令集才能对栈中的数据进行逻辑运算。

同时，栈容器一定与 Java 函数相关联，或者说 JVM 必须为每一个 Java 函数划分一定的空间作为栈式指令集操作的对象，而本来每一个 Java 函数都有一个与之配套的栈帧空间，所以 JVM 干脆将函数的栈帧空间与这个操作栈合二为一，以免另外再维护一套与堆栈类似的内存空间。JVM 具体的做法是将操作栈直接嵌入到 Java 方法的栈帧之中，作为 Java 方法栈帧的一部分。

如此一来，Java 方法栈就包含了至少 3 部分数据：

- ◎ Java 方法的局部变量表。
- ◎ Java 方法堆栈调用的上下文环境数据。
- ◎ Java 方法的操作数栈。

事实上，Java 方法栈帧的结构的确与上文所推导出来的结构保持一致。一个比较详细的 Java 方法栈帧如图 7.16 所示（从下往上看）。

在编译程序代码时，栈帧中需要多大的局部变量表、多深的操作数栈都已经完全确定，并且写入到了字节码文件中方法表的 `Code` 属性之中。因此，一个栈需要分配多少内存，不会受到程序运行期变量数据的影响，而仅仅取决于具体的虚拟机实现。在这一点上，Java 语言与 C/C++ 等语言一样，都是在编译期计算好所需栈帧大小。

在这里需要提一下“动态链接”的概念。在 C/C++ 语言中有动态链接库的概念，Java 中也有类似的概念。每个栈帧都包含一个指向运行时常量池中该栈帧所属方法的引用，持有这个引用是为了支持方法调用过程中的动态链接。class 文件的常量池中有大量的符号引用，字节码中的方法调用指令就以常量池中指向方法的符号引用为参数。这些符号引用一部分会在类加载阶

段或第一次使用时转化为直接引用，这种转化称为静态解析。另一部分将在每一次的运行期间转化为直接应用，这部分称为动态链接。因此要实现动态链接的关键便是在 Java 方法栈中持有一个指针，指向常量池，如此便能得到该 Java 方法的字节码指令，并根据字节码指令映射到机器指令，从而完成方法逻辑处理。

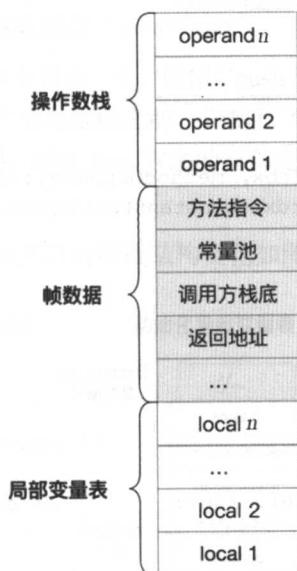


图 7.16 Java 栈帧结构图

7.4.2 栈帧创建

HotSpot 中生成 JVM 栈帧的代码如下：

清单：/src/cpu/x86/vm/templateInterpreter_x86_32.cpp

作用：JVM 创建栈帧

```
void TemplateInterpreterGenerator::generate_fixed_frame(bool native_call) {
    // initialize fixed part of activation frame
    __ push(rax);                                // save return address
    __ enter();                                  // save old & set new rbp,

    __ push(rsi);                                // set sender sp
    __ push((int32_t) NULL_WORD);                // leave last_sp as null
    __ movptr(rsi, Address(rbx, methodOopDesc::const_offset())); // get constMethodOop
    __ lea(rsi, Address(rsi, constMethodOopDesc::codes_offset())); // get codebase
    __ push(rbx);                                // save methodOop
```

```

    if (ProfileInterpreter) {
        Label method_data_continue;
        __ movptr(rdx, Address(rbx, in_bytes(methodOopDesc::method_data_offset())));
        __ testptr(rdx, rdx);
        __ jcc(Assembler::zero, method_data_continue);
        __ addptr(rdx, in_bytes(methodDataOopDesc::data_offset()));
        __ bind(method_data_continue);
        __ push(rdx); // set the mdp (method data pointer)
    } else {
        __ push(0);
    }

    __ movptr(rdx, Address(rbx, methodOopDesc::constants_offset()));
    __ movptr(rdx, Address(rdx, constantPoolOopDesc::cache_offset_in_bytes()));
    __ push(rdx); // set constant pool cache
    __ push(rdi); // set locals pointer
    if (native_call) {
        __ push(0); // no bcp
    } else {
        __ push(rsi); // set bcp (byte code pointer)
    }
    __ push(0); // reserve word for pointer to
expression stack bottom
    __ movptr(Address(rsp, 0), rsp); // set expression stack bottom
}

```

这个过程大体上可以分为以下几步：

- (1) 恢复 return address。
- (2) 创建新的栈帧。
- (3) 将最后一个人参位置压栈。
- (4) 计算 Java 方法的第一个字节码位置。
- (5) 将 methodOop 压栈。
- (6) 将 ConstantPoolCache 压栈。
- (7) 将局部变量表压栈。
- (8) 将第一条字节码指令压栈。
- (9) 将操作数栈栈底地址压栈。

下面开始详细讲解这几个步骤。如果你将这几个步骤的技术实现都研究清楚了，那么你便会真正明白 Java 方法栈帧创建的机制。

1. 恢复 return address

JVM 创建栈帧的第一步是恢复 return address。所谓 return address 的概念，本书前面的章节多次描述过，就是 eip 寄存器，也是 JVM 调用目标 Java 方法的 call 指令的下一条指令的内存地址。

这一步在 TemplateInterpreterGenerator::generate_fixed_frame(bool native_call) 函数中所对应的代码是 “--push(rax),” 其对应的汇编是 “push %eax”。

在上面的步骤——“创建局部变量表”中，使用 “push \$0x0” 这样的指令连续压栈，将 Java 方法内部的局部变量压入堆栈，但是在压栈之前，先执行了 “pop %eax” 这条指令，将 return address 从栈顶弹出至 eax 寄存器中。现在 Java 方法的局部变量全部入栈，于是又将 return address 再次还原到堆栈中。

Java 方法局部变量入栈之前和之后的堆栈结构变化如图 7.17 所示。



图 7.17 局部变量压栈之前和之后的堆栈对比图

由于 Java 方法的人参和 Java 方法内部的局部变量共同组成了局部变量表,局部变量表作为一个整体,自然不能被分隔,不能硬生生地在中间插入一个 return address,因此在对局部变量进行压栈时必然要先将 return address 拿掉,待局部变量全部压栈完成之后,再将 return address 恢复至栈顶。

2. 开辟新的栈帧

这里所谓创建新的栈帧,是指通过硬件寄存器真正开始为被调用的 Java 方法分配堆栈空间。创建栈帧的第一步就是首先要清晰地标识出自己的栈底,栈底就是一个新的栈帧的领域边界。如果没有标识出栈底,则不管分配多少新的堆栈空间,都还只是在别人的栈帧领域之内。这一点与动物世界一样,狗在路边撒泡尿就对外宣称这是自己的领域,别的动物不能再霸占,否则就要发生“世界大战”。栈帧划分自己领域的方法也很简单,执行下面这 2 条机器指令即可:

◎ push %ebp

◎ mov %esp,%ebp

从这里开始,被调用的 Java 方法终于开始有了自己栈帧的“立脚之处”,接下来新增的堆栈空间就都属于被调用的 Java 方法,而不再是“调用方法”。以后对这块区域内的数据进行访问时,都必须以新的 bp(栈底指针)为基准进行偏移寻址,而不能以调用方的 bp 为基准进行偏移寻址。

不过刚才“从这里开始,被调用的 Java 方法终于开始有了自己栈帧的立脚之处”这种说法却不够严谨,因为被调用的 Java 方法并不是从这里才开始拥有自己的栈帧空间的,在前面的步骤——创建局部变量表时,所创建的局部变量表其实正是被调用的 Java 方法自己的栈帧空间的一部分,所以说从那时候起,被调用的 Java 方法就拥有了自己的栈帧领域,而并不是执行了本步骤中的“push %ebp”和“mov %esp,%ebp”这两条指令之后才开始拥有自己的栈帧领域。这一点与 C/C++等语言有巨大的差异,在 C/C++中,大部分情况下,调用了这两条机器指令后,接下来会再配合调用“sub \$operand,%esp”为被调用的函数分配新的栈帧,因此在 C/C++中,这两条机器指令的出现,往往就意味着被调用的函数将会开辟自己的栈帧空间。而在 Hotspot 中,当这两条机器指令出现时,被调用的 Java 方法已经拥有自己的栈帧领域了。

不过话说回来,既然局部变量表本身就属于被调用的 Java 方法的栈帧的一部分,那么 HotSpot 应该在创建局部变量表之前就执行“push %ebp”和“mov %esp,%ebp”这两条机器指令,为被调用的 Java 方法创建栈帧。不过 HotSpot 一反常规,没有这么做,这里面的原因倒也不复杂,主要基于下面两点考虑。

1) 局部变量表的整体性

在本步骤之前有一个步骤——“创建局部变量表”，这个步骤被安排在 `entry_point` 例程里执行，但是严格来说，这个步骤并不能叫作“创建局部变量表”，而只能叫作将 Java 方法局部变量压栈，这是因为局部变量表不仅仅包含 Java 方法的局部变量，而且包含 Java 方法的入参，如图 7.18 所示。

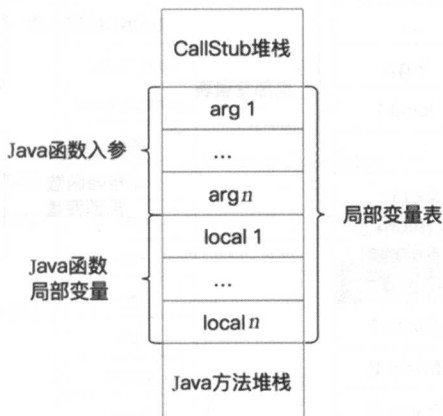


图 7.18 Java 方法的局部变量表覆盖范围

Java 函数入栈逻辑在 `entry_point` 例程的调用方——`CallStub` 例程中执行，而 Java 函数局部变量入栈逻辑则在 `entry_point` 例程中完成，虽然 Java 函数入参的堆栈区域与 Java 函数局部变量所在的堆栈区域分属于两个不同的栈帧（前者属于 `CallStub` 的栈帧，而后者按理说属于 `entry_point` 的栈帧），但是对于被调用的 Java 函数而言，不管是其方法入参还是方法内部的局部变量，都隶属于局部变量表，当 Java 程序执行 Java 字节码指令读写 Java 方法的局部变量表时，将 Java 函数的第一个入参所在的堆栈位置作为偏移基址对局部变量进行变址寻址。而调用函数的栈底基址 `ebp` 很明显并不属于被调用的 Java 方法的局部变量表的一员，自然不能被字节码读写，因此 HotSpot 并不能在对局部变量进行压栈之前执行“`push %ebp`”和“`mov %esp, %ebp`”这两条机器指令，否则局部变量表就不完整了。这与前面步骤先将 `return address` 出栈弹出到 `rsa`、等到将 Java 方法的局部变量全部入栈之后再恢复至栈顶是同一个道理。

2) Java 栈帧数据相对寻址

在这一步才执行“`push %ebp`”和“`mov %esp, %ebp`”这两条机器指令的另一个重要原因是，为了对 Java 方法栈帧内部的帧数据（`fixed frame`）进行相对寻址方便。上文讲过，Java 方法的栈帧主要由 3 部分组成：局部变量表、帧数据和操作数栈。其中只有帧数据的这部分数据的长度是固定不变的，任何 Java 方法的这部分内容所占用的堆栈内存空间大小相等，因此在 HotSpot 内部，这部分数据被称作“`fixed frame`”。而局部变量表与操作数栈的大小则随着 Java 方法的不

同而变化，并没有固定的大小。看图 7.19 所示的左图和右图。

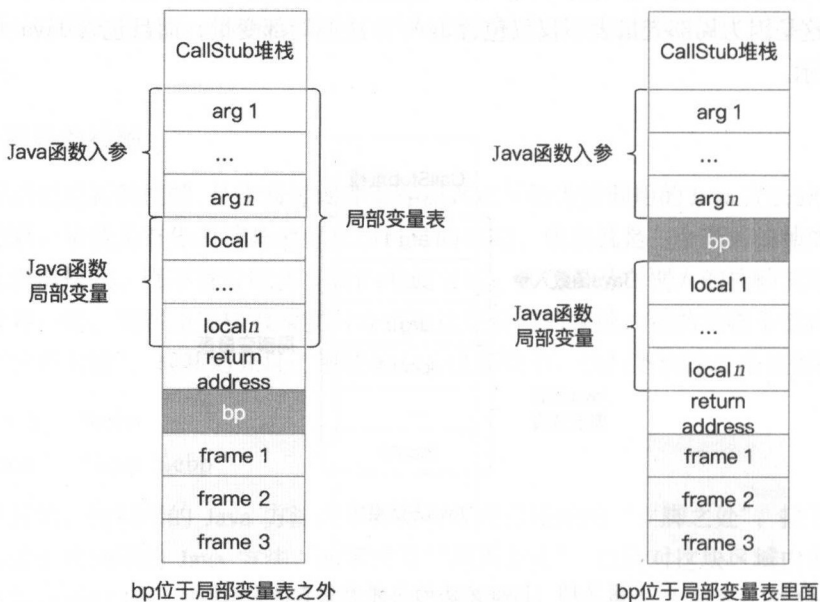


图 7.19 bp 位于局部变量表不同位置处的堆栈示意图

在图 7.19 的左图中，bp 是 Hotspot 实际所压栈的位置，位于局部变量表顶部；而右图中 bp 则位于局部变量表内部、Java 方法入参顶部、局部变量区域底部。示意图中的 frame 1、frame 2、frame 3 则是 Java 方法栈帧中的 fixed frame 部分的 3 个示例数据。在 Hotspot 执行 Java 方法的过程中，经常需要读取堆栈中的 fixed frame 部分的数据，而读取的方式便是基于 bp 进行变址寻址。以读取 frame 1 这个数据为例，在左图中，frame 1 与 bp 相邻，因此 Hotspot 要读取 frame 1，只需将 bp 减 4 即可（假设一个数据占用 4 字节空间）。同理，读取 frame 2 和 frame 3 只需分别将 bp 减去 8 和 12。但是对于右图，由于 bp 与 frame 1 之间隔了一个局部变量区域，而不同的 Java 方法，其局部变量的数量相差很大，因此要通过 bp 去寻址 frame 1，每次都得计算出 Java 方法的局部变量所占的内存总和，这将十分消耗 Java 虚拟机的性能。即使将每个 Java 方法的局部变量表所占的内存空间存储起来也于事无补，那样的话仍然要读取内存。

不过虽然 bp 被放在了被调用的 Java 方法的局部变量区域的后面，但是局部变量仍然应当被看作是被调用的 Java 方法的栈帧空间的一部分，毕竟 Java 方法的字节码在访问方法自己的局部变量表时，是将局部变量表当作自己堆栈空间的一部分，而不是别人的堆栈空间。

“push %ebp”和“mov %esp, %ebp”这两条机器指令执行完成之后的堆栈内存空间布局如图 7.20 所示。

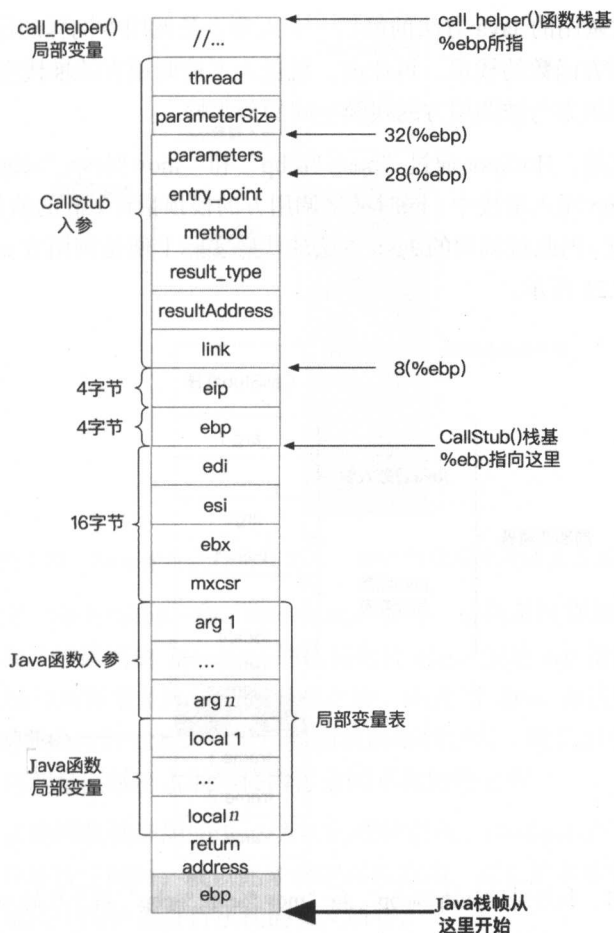


图 7.20 Java 方法栈帧刚开始的堆栈布局

3. 将最后一个入参位置压栈

TemplateInterpreterGenerator::generate_fixed_frame(bool native_call) 函数接着执行 `__push(rsi)` 指令，其对应的机器指令是“`push %rsi`”。在前面讲解 `CallStub()` 函数时，当时 `rsi` 寄存器里保存了 Java 方法最后一个入参在堆栈中的位置，所以执行“`push %rsi`”之后，Java 方法最后一个入参的位置就被压入 Java 的方法栈中。

`entry_point` 例程由 `CallStub` 例程调用，也可能会由其他例程调用，例如 Java 方法调用 Java 方法时，就会由 `invoke_virtual` 或者 `invoke_special` 等例程调用。当 `entry_point` 例程由 `CallStub` 例程调用时，`CallStub` 例程流程进入 `entry_point` 时，其栈顶元素正好就是被调用的 Java 方法的最后一个入参，所以这个位置就是 `CallStub` 例程所对应的函数的栈顶。同样，当 `entry_point` 由

其他例程进入时，被调用的 Java 方法的最后一个人参也是调用方例程的栈顶。所以，这最后一个人参自然就是调用方函数的栈顶，再往前，就进入了被调用方的堆栈空间，因此最后一个参数的位置自然成为调用方与被调用方的栈帧空间的分水岭。

在执行本指令之前，HotSpot 通过“push %ebp”和“mov %esp, %ebp”这两条机器指令将调用方的栈底指针 ebp 压入堆栈中，同时又将调用方的栈顶指针 esp 的值复制给 ebp，作为被调用的 Java 方法的栈底，因此被调用的 Java 方法的栈底实际上便是调用方 esp 栈顶指针，此时 esp 所指向的位置如图 7.21 所示。

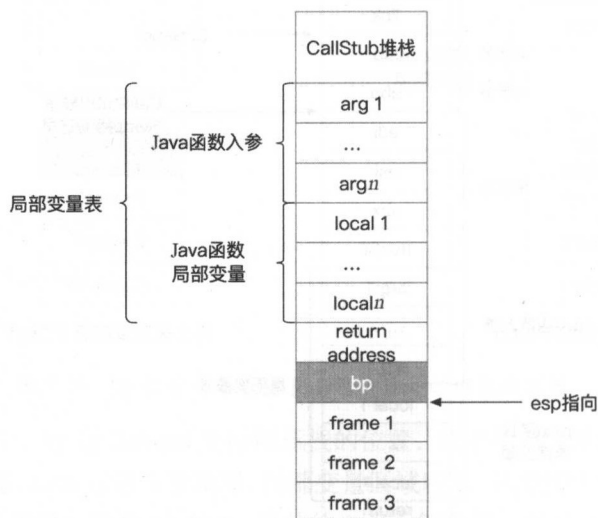


图 7.21 执行完“push %ebp”和“mov %esp, %ebp”指令后的 esp 位置

此时 esp 实际上指向了 Java 方法栈帧“fixed frame”的底部，这个位置已经超出了 Java 方法局部变量区域，这对于 Java 方法的运行倒没有任何影响，但是等 Java 运行完成，即将要退出时，就会引起很大的问题。

当 Java 程序运行完成时，HotSpot 必须回收其对应的堆栈空间。对于操作系统而言，堆栈空间的回收别无他法，全都依赖 esp 寄存器值的修改。对于 C/C++等本地编译型的语言，ebp 指针指向被调用函数栈帧的栈底，因此回收被调用函数的栈帧空间时，只需将 esp 恢复至被调用函数的 ebp 指针即可。Java 方法栈帧由局部变量表、帧数据和操作数栈这三部分所组成，其中局部变量表又一分为二，一部分为 Java 方法入参区域，一部分则为 Java 方法局部变量区域，前者属于调用方的栈帧空间，后者才属于被调用一方的栈帧空间，因此当被调用的 Java 方法执行完毕后，HotSpot 需要回收的堆栈空间实际上仅包含帧数据、操作数栈及局部变量表中的局部变量所在的区域，如图 7.22 所示。

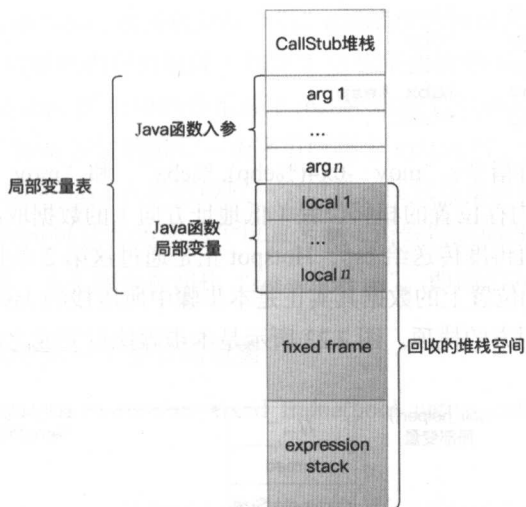


图 7.22 Java 方法执行完成之后，回收的堆栈空间涵盖范围

在上一步，执行完“push %ebp”和“mov %esp, %ebp”这两条机器指令后，调用方函数的栈顶指针 esp 被复制给了被调用的 Java 方法的栈底指针 ebp，但是 esp 指向的位置并不是 Java 局部变量表中入参区域与局部变量区域的分界线之处，因此当 Java 方法执行完成之后，如果 Hotspot 以 Java 方法的栈底指针 ebp 来确定所要回收的堆栈空间，就会出现异常，因为 Java 方法局部变量表中的局部变量区域所占的堆栈内存空间不在回收之列。

因此，为了能够正确回收被调用的 Java 方法的栈帧空间，HotSpot 必须记录 Java 方法的调用方的栈顶位置，回收堆栈空间时，就以此来确定回收范围。这便是本步骤——将 Java 方法入参的最后一个参数在堆栈中的位置进行压栈的意义所在。

事实上，如果将 Java 字节码中的 return 指令展开成机器指令，也的确会发现其中的奥妙。假设一个 Java 方法的返回值类型是 int，则该 Java 方法的最后一条字节码指令必定是 ireturn，ireturn 指令展开后的机器码如下所示（使用汇编助记符展示）：

```
ireturn 172 ireturn [0xb36dbca0, 0xb36dbec0] 544 bytes
```

```
[Disassembling for mach='i386']
```

```
0xb36dbca0: pop    %eax
0xb36dbca1: mov    %esp, %ecx
0xb36dbca3: shr    $0xc, %ecx
0xb36dbca6: mov    -0x48f07d20(, %ecx, 4), %ecx
//.....
```

```
0xb36dbe95: add    $0x8, %esp
0xb36dbe98: pop    %eax
0xb36dbe99: mov    -0x4(%ebp), %ebx
```

```
0xb36dbe9c: mov    %ebp,%esp
0xb36dbe9e: pop    %ebp
0xb36dbe9f: pop    %esi
0xb36dbea0: mov    %ebx,%esp
0xb36dbea2: jmp    *%esi
```

观察其中加粗的两行指令：“mov -0x4(%ebp), %ebx” 和 “mov %ebx, %esp”，第一条指令将 `ebp` 所指向的堆栈内存位置的相邻位置（低地址方向）的数据取出来，传送给 `ebp`；接着第二条指令便将 `ebp` 的值再度传送给 `esp`，Hotspot 正是通过这第 2 条指令彻底回收了当前 Java 方法的栈帧。-0x4(%ebp)位置上的数据其实正是本步骤中所压栈的 Java 方法的最后一个入参在堆栈中的位置，也即调用方的栈顶。图 7.23 所示是本步骤执行完成之后的堆栈内存布局。

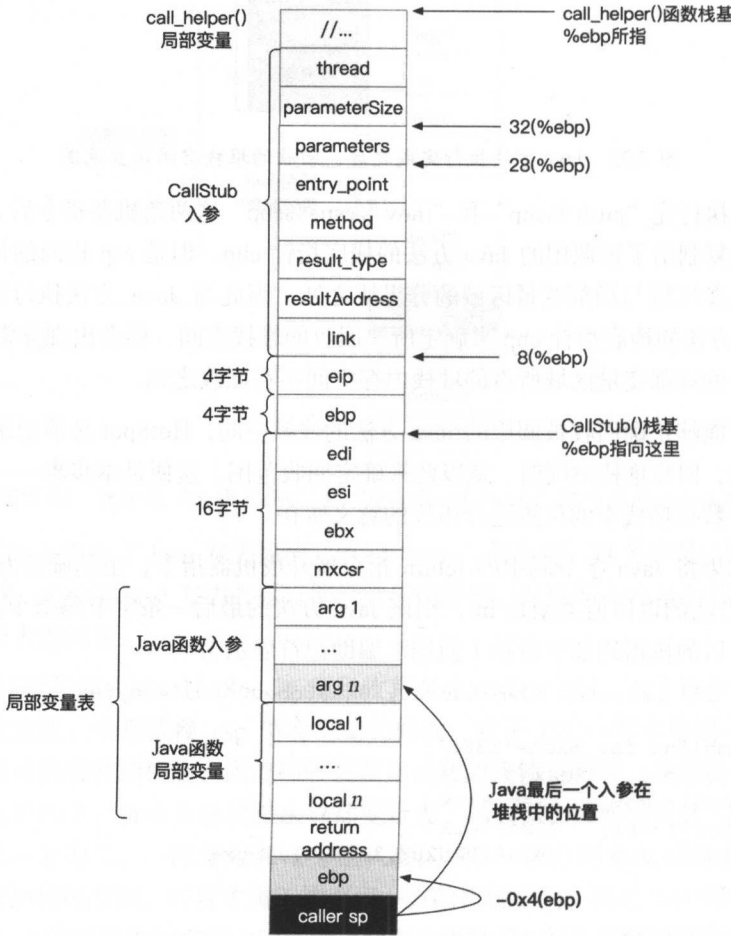


图 7.23 执行完 “push %rsi” 指令之后的堆栈空间布局图

执行完本步骤的“push %rsi”这条机器指令后，caller sp 就被压入栈中，这个位置中所存储的值其实就是 arg n 所在的堆栈内存的地址（如图 7.23 中箭头所指）。同时，紧邻 caller sp 上方的位置中所存储的值正是 ebp，在 32 位操作系统上，caller sp 相对于 ebp 的位置就是 $-0x4(\%ebp)$ ，因此最终 Hotspot 在执行 Java 方法的最后一条字节码指令 ireturn 时，通过 $-0x4(\%ebp)$ 就能获取 caller sp，而 caller sp 又指向 Java 入参的最后一个位置，此位置也恰好是 Java 方法的调用方的栈顶，注意，栈顶哟！Hotspot 正是依赖这个位置来确定需要回收的堆栈内存的界限的。

所以在 HotSpot 内部，本步骤也被叫作“set sender sp”，即调用方栈顶保存。

4. 计算 Java 方法的第一个字节码位置

TemplateInterpreterGenerator::generate_fixed_frame(bool native_call)函数接着执行下面两行代码：

```

◎ __ movptr(rsi, Address(rbx,methodOopDesc::const_offset()));
  // get constMethodOop
◎ __ lea(rsi, Address(rsi,constMethodOopDesc::codes_offset()));
  // get codebase

```

这两行代码所对应的机器码是：

- ◎ mov 0x8(%ebx),%esi。ebx 指向 method，0x8(%ebx)指向 ConstMethod。
- ◎ lea 0x30(%esi),%esi。在这一步让%esi 指向 codebase，也就是 method 的第一个字节码的位置。

这一段不难理解，在进入 entry_point 例程之前，ebx 寄存器中所保存的就是 Java 方法在 JVM 内部对应的 methodOop 对象首地址。0x8(%ebx)恰好指向 constMethodOop（method 与 constMethod 这两个对象会在下一章详细讲解），constMethodOop 对象在常量池解析阶段生成，该对象主要保存 Java 方法中的只读信息，例如异常信息表、Java 方法注解信息、方法名、Java 方法的字节码指令等。

methodOop 的内存结构如下（JDK 6，32 位 x86 Linux 平台）：

```

methodOop
+0:  header
+4:  klass
+8:  constMethodOop
+12: constants
+16: methodData
//...

```

由此可见，相对 methodOop 首地址偏移 8 字节的位置处所保存的正是指向 constMethodOop

对象的指针，因此 HotSpot 执行 “mov 0x8(%ebx), %esi” 这条指令之后，esi 寄存器中所存储的就是 constMethodOop 对象的内存地址了。

这里为了描述方便（免得众位道友辛苦翻看源码），简单贴出 constMethodOop 的布局结构与各字段偏移量（JDK 6，32 位 x86 Linux 平台）：

```
constMethodOop
+0:    header
+4:    klass
+8:    fingerprint
+16:   is_conc_safe
+20:   method
+24:   stackmap_data
+28:   exception_table
+32:   constMethod_size
+36:   interpreter_kind
+37:   flags
+38:   code_size
+40:   name_index
+42:   signature_index
+44:   method_idnum
+46:   generic_signature_index
+48:   byte codes
```

注意，在这个结构中，is_conc_safe 是 bool 类型，仅占用 1 字节。但是其后面紧跟的字段是 methodOop 指针类型，其宽度是 4，因此需要按 4 字节对齐，也因此 is_conc_safe 后面有 3 字节的补位。

前文在分析 Hotspot 解析 Java 类的方法时，会将 Java 类方法的字节码指令保存到对应的 constMethodOop 对象实例的末尾。constMethodOop 类型的最后一个字段是 generic_signature_index，类型是 u2，其相对于 constMethodOop 对象首地址的偏移量是 46，而 Java 方法的字节码指令就分配在该字段的后面，因此 Java 方法的字节码的第一条指令的位置相对于 constMethodOop 就是 48。因此，在本步骤会通过 “lea 0x30(%esi), %esi” 指令将 Java 方法的第一条字节码指令的内存位置传送给 esi 寄存器。在 entry_point 例程的最后，会通过 call 指令去开始真正执行 Java 方法的字节码指令。

5. 将 methodOop 压栈

TemplateInterpreterGenerator::generate_fixed_frame(bool native_call)函数接着执行下面这行代码：

```
◎  __ push(rbx)    // save methodOop
```

其对应的机器指令是：

```
◎ push %rbx
```

rbx 寄存器指向 Java 方法在 JVM 内部所对应的 methodOop 对象首地址,因此这一步的目的便是将 methodOop 对象的首地址压入栈中。在 Hotspot 调用 Java 方法的过程中,将通过这个地址读取到 Java 方法的全部信息,例如进行多态运行期动态方法绑定时需要定位到 callee 从而决定到底调用继承体系中的哪一个对象的方法。

6. 将 ConstantPoolCache 压栈

TemplateInterpreterGenerator::generate_fixed_frame(bool native_call)函数接着执行下面这几行代码：

```
◎ __ movptr(rdx, Address(rbx, methodOopDesc::constants_offset()));
◎ __ movptr(rdx, Address(rdx, constantPoolOopDesc::cache_offset_in_bytes()));
◎ __ push(rdx); // set constant pool cache
```

其对应的机器指令是：

```
◎ mov0xc(%ebx),%edx --ConstMethod *
◎ mov0xc(%edx),%edx --ConstantPoolCache *
◎ push%edx
```

在 32 位 x86 Linux 平台上,相对于 methodOop 首地址偏移量为 0xc (即十进制 12) 的字段是 constantPoolOop 指针,该指针指向 Java 类所对应的内存常量池首地址,Hotspot 通过 “mov 0xc(%ebx),%edx” 指令将 constantPoolOop 首地址传送给 edx 寄存器。

而 constantPoolOop 的内存结构布局如下所示 (前文详细描述过,此处简略展示一二):

```
constantPoolOop
+0: header
+4: klass
+8: tags
+12: constantPoolCacheOop
+16: _pool_holder
//...
```

相对于 constantPoolOop 首地址偏移量为 0xc 的字段是 constantPoolCacheOop, HotSpot 通过 “mov 0xc(%edx),%edx” 指令将 constantPoolCacheOop 首地址传送给 edx 寄存器,接着执

行“push %edx”指令将 constantPoolCacheOop 首地址压入 Java 方法栈中。由此可知，对于同一个 Java 类文件中的所有 Java 方法，每一个 Java 方法的栈帧中都必须持有指向该 Java 类解析后所生成的常量池缓存对象的地址。常量池缓存中的内容皆是直接引用，不必像常量池那样，存放的都是索引号。

到了这一步，堆栈内存布局结构如图 7.24 所示。

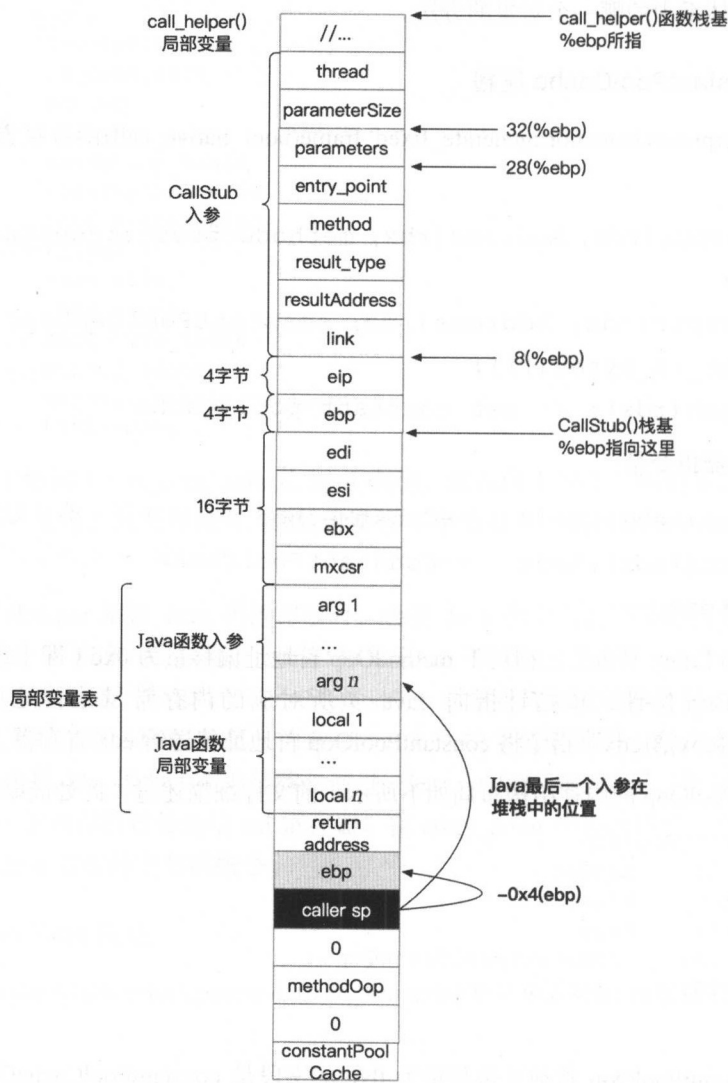


图 7.24 constantPoolCacheOop 入栈后的堆栈内存布局结构

注意，在靠近栈顶的位置处多出了 2 个 0，这两个位置为保留字段或者占位符，在 Java 方法运行过程中将会用于存储运行期的部分结果。

这里需要说明的是，这种堆栈内存布局图绘制到现在已经变得很大、很复杂了，笔者本想砍去一部分，但是最终仍然保留了全部信息。因为本书浓墨重彩地描述了 Java 方法调用的入口及栈帧创建的全过程，从 JVM 内部的 CallStub 例程开始，一直到现在，中间横跨各种技术点，但是不管中间补充了哪些侧面的信息，主线一直未变，这张图就是主线。相信各位道友如果是从开始一路读下来，只要看到这张图，自然会感到主线清晰，并不会因为中间其他技术点的插入而扰乱了思路。所以虽然这张图变得十分复杂，但笔者丝毫不敢精简一分半点，不然可能连笔者自己写着写着都可能会写偏了。

7. 将局部变量表压栈

TemplateInterpreterGenerator::generate_fixed_frame(bool native_call)函数接着执行 `_push(rdi)` 指令，其对应的机器指令是 `push %edi`。

在前面初始化 Java 方法局部变量区域时，通过不断地循环执行“`push $0x0`”指令为 Java 方法内的局部变量开辟堆栈空间，而在循环之前，将 Java 方法的第一个入参在堆栈中的位置传送给 `edi` 寄存器，并且中间并没有被修改，因此此时 `edi` 寄存器仍然指向 Java 方法的第一个入参在堆栈中的位置。

这一步对于 Java 方法而言具有十分重要的意义，因为 Java 方法栈帧部分的局部变量表的起始位置，其实就是 Java 方法的第一个入参在堆栈中的位置，因此 Java 栈帧必须要记录下这个位置，作为 Java 方法的局部变量表的第一个槽位，不然在 Java 方法执行的过程中，Java 字节码无法读写局部变量表。

举一个十分简单的例子，有如下 Java 类：

清单：A.java

作用：Java 方法局部变量表

```
class A{
    public static void doSomething(){
        int a = 3;
        int b = 81;
        int c = a + b - 9;
    }
}
```

使用 `javap -verbose` 命令查看 `doSomething()` 方法所对应的字节码如下：

```
public static void doSomething();
descriptor: ()V
```

```

flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=3, args_size=0
    0: iconst_3
    1: istore_0
    2: bipush      81
    4: istore_1
    5: iload_0
    6: iload_1
    7: iadd
    8: bipush      9
   10: isub
   11: istore_2
   12: return

```

当 Hotspot 执行源代码中的 `int a = 3` 时，实际上执行的是 `iconst_3` 和 `istore_0` 这两条字节码指令，其中 `iconst_3` 字节码所对应的汇编指令如下：

```

iconst_3 6 iconst_3 [0xb36d7ce0, 0xb36d7d20] 64 bytes
[Disassembling for mach='i386']
0xb36d7d04: mov    $0x3,%eax //将操作数 3 推送至栈顶缓存（缓存在 eax 寄存器中）
0xb36d7d09: movzbl 0x1(%esi),%ebx
0xb36d7d0d: inc    %esi
0xb36d7d0e: jmp    *-0x48f106a0(,%ebx,4)

```

当 Hotspot 执行完 `iconst_3` 字节码指令后，操作数 3 就被推送到 `eax` 寄存器中（这里涉及栈顶缓存，下文会详细讲解栈顶缓存的概念及实现机制）。接着在 `istore_0` 字节码指令中理应读取栈顶缓存的值，并将其保存到 `doSomething()` 方法的局部变量表的第一个槽位上（因为变量 `a` 是 `doSomething()` 方法内的第一个局部变量）。来看 `istore_0` 所对应的汇编指令：

```

istore_0 59 istore_0 [0xb36d9240, 0xb36d9260] 32 bytes
[Disassembling for mach='i386']
0xb36d9241: mov    %eax, (%edi) //这里引用了 edi 寄存器
0xb36d9243: movzbl 0x1(%esi),%ebx
0xb36d9247: inc    %esi
0xb36d9248: jmp    *-0x48f0f2a0(,%ebx,4)

```

在 `istore_0` 字节码所对应的汇编指令中，执行 “`mov %eax, (%edi)`” 指令，将栈顶缓存的值（即 `eax` 寄存器中所保存的值）传送给 `edi` 寄存器所指向的内存位置。而 `edi` 寄存器正指向 Java 局部变量表的第一个槽位，而该槽位其实也是 Java 方法的第一个入参所在的堆栈内存位置。只不过本示例中的 `doSomething()` 方法没有入参（静态方法，连隐藏的 `this` 指针也没有），因此 `edi` 寄存器的位置只能是 Java 方法内的第一个局部变量的堆栈位置。

在本例中，`doSomething()` 函数的第二句源代码是 `int b = 81`，变量 `b` 是 `doSomething()` 方法内的第 2 个局部变量，那么在 32 位 x86 平台上，其位置应该是在 `doSomething()` 方法的局部变

量表的第 2 个槽位, 而 `edi` 寄存器指向第 1 个槽位, 因此第 2 个槽位相对于第一个槽位的偏移量是 $-0x4$, 使用 `edi` 表示应该是 $-0x4(\%edi)$ 。看看 `int b = 81` 所对应的字节码指令是不是这么处理的就知道了。`int b = 81` 对应两条字节码指令:

◎ `bipush 81`

◎ `istore_1`

其中, `bipush 81` 也会将 81 这个操作数推送至栈顶缓存, 保存在 `eax` 寄存器中。`istore_1` 字节码指令的功能便是将栈顶缓存的内容保存到 Java 方法局部变量表的第 2 个槽位上, 一起围观其对应的机器指令:

```
istore_1 60 istore_1 [0xb36d9280, 0xb36d92a0] 32 bytes
[Disassembling for mach='i386']
0xb36d9281: mov    %eax, -0x4(%edi)
0xb36d9284: movzbl 0x1(%esi), %ebx
0xb36d9288: inc    %esi
0xb36d9289: jmp    *-0x48f0f2a0(, %ebx, 4)
```

这条字节码指令果然引用了 $-0x4(\%edi)$ 这个位置, 通过 `edi` 寄存器直接定位到 Java 方法的第 2 个槽位。

总之, 在 Java 方法运行期间, 对 Java 方法入参和内部局部变量的读写就全靠 `edi` 寄存器了。这么一个重要的数据当然要保存到 Java 方法的栈帧里去了。

`edi` 在 Hotspot 内部也被叫作 `locals pointer`, 即指向局部变量表的指针。

8. 将第一条字节码指令压栈

`TemplateInterpreterGenerator::generate_fixed_frame(bool native_call)` 函数接着执行的源代码是:

```
if (native_call) {
    __ push(0);    // no bcp
} else {
    __ push(rsi);  // set bcp(byte code pointer)
}
```

由于 JVM 既可以加载 Java 类, 也可以加载 C/C++/Delphi 等程序库, 因此 Hotspot 需要通过 `native_call` 判断即将被调用的方法是否是 Java 方法。如果是 Java 方法, 就需要将 Java 方法所对应的第一个字节码指令的位置入栈, 这个位置在前面的步骤中已经计算好, 并被保存在 `rsi` 寄存器中, 因此只需执行 “`push %rsi`” 指令即可。反之, 如果被调用的方法不是 Java 方法, 那就是本地方法。C/C++/Delphi 等程序库都会被 JVM 当作本地方法调用。所谓本地方法, 也即直接本地编译型方法, 例如 C 语言程序, 编译后直接生成能够被 CPU 识别的二进制机器指令,

所以本地方法不存在所谓“Java 字节码指令”一说，因此也就无须将 rsi 寄存器压入栈中。

rsi 所指向的位置在 Hotspot 内部有一个专门的称呼——bcp，其含义是 byte code point，即指向 Java 方法字节码指令的指针。

9. 将操作数栈栈底地址压栈

TemplateInterpreterGenerator::generate_fixed_frame(bool native_call)函数所执行的最后两句源代码如下：

- ◎ `__ push(0);`
- ◎ `__ movptr(Address(rsp, 0), rsp)`

其对应的机器指令如下：

- ◎ `push $0x0`
- ◎ `mov %esp, (%esp)`

第一条机器指令往栈顶压入一个零值，接着通过第二条机器指令将当前 esp 寄存器的值覆盖为刚刚压入的零值。

在前面步骤中执行各种 push 操作，每一次 push 完之后，物理 CPU 会自动将 esp 寄存器的值更新为当前最新的栈顶位置，因此到了本步执行的时候，esp 寄存器本来就指向了当前堆栈的栈顶，而这里又通过“mov %esp, (%esp)”指令将 esp 的值覆盖为刚刚压入栈顶的零值，因此此时栈顶所保存的值就是其自己的内存位置。此时堆栈内存布局结构如图 7.25 所示。

在这一步，之所以要将 esp 的值存储起来，是因为截止到本步骤，Java 方法栈帧的“fixed frame”部分就创建完成了，Java 方法栈帧接下来的部分就是操作数栈。虽然操作数栈也属于 Java 方法栈帧的一部分，但是在 Java 方法的运行过程中，Java 字节码指令所面向的栈，实际上便是这个“操作数栈”，而不是整个 Java 方法栈帧。Java 字节码进行压栈和出栈，会基于操作数栈的栈底位置进行变址寻址。由于 Java 方法的 fixed frame 接下来相邻的部分就是操作数栈，因此 fixed frame 的栈顶位置其实就是操作数栈的栈底，HotSpot 将该位置记录下来，用作操作数栈的栈底，在 Hotspot 内部，将该位置叫作“expression stack bottom”，即表达式栈栈底。

在这一步之所以要将 esp 寄存器的值存储起来的另一个原因是，Java 字节码的压栈与出栈指令，最终都会映射成物理机器码的 push 和 pop 指令，而这两条指令都会自动修改 esp 的值。而当 Hotspot 执行到这一步时，esp 恰好指向了 fixed frame 的顶部，因此 Hotspot 将其保存起来，一方面是为了能够定位到 fixed frame 的顶部位置，另一方面却也是方便了 Java 字节码指令的压栈与出栈的执行。

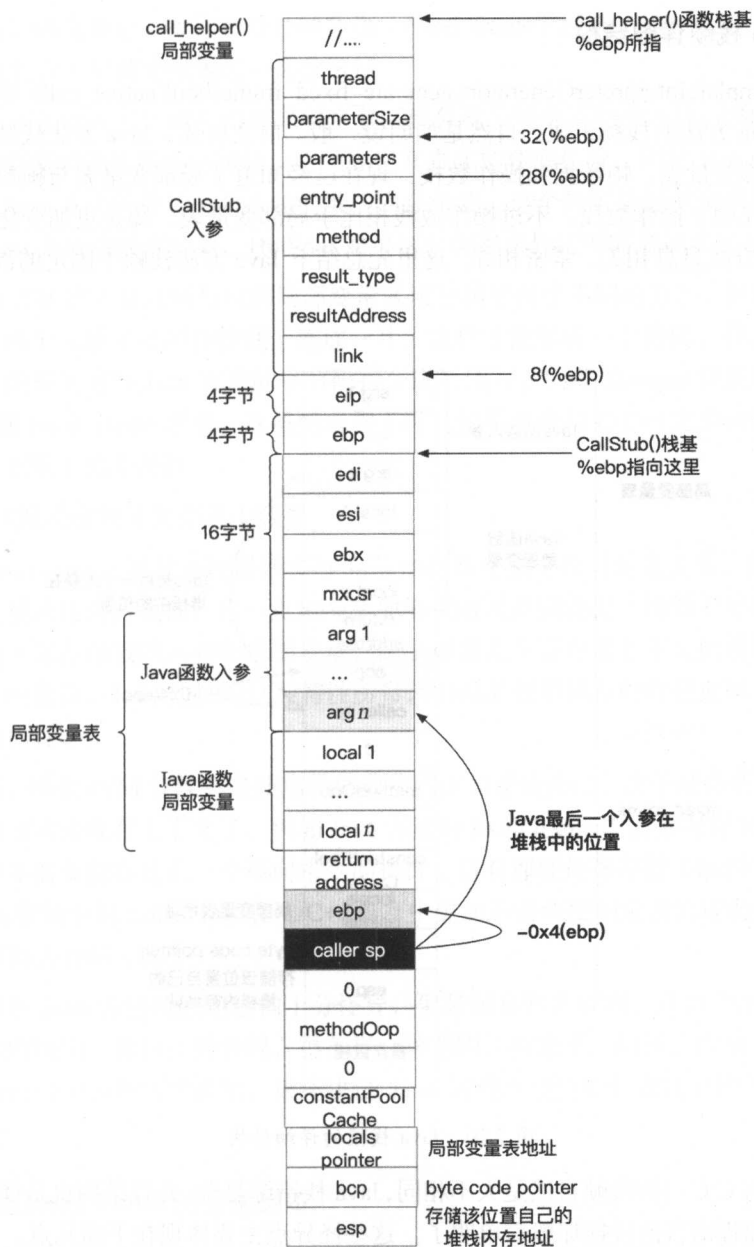


图 7.25 TemplateInterpreterGenerator::generate_fixed_frame(bool native_call)

函数执行完之后的堆栈内存结构布局

10. Java 栈帧详细结构

分析完 `TemplateInterpreterGenerator::generate_fixed_frame(bool native_call)` 这个函数，现在回头再看看 Java 方法的栈帧组成，自然是如明镜一般。前文所述，Java 方法栈帧由 3 大部分组成，分别是局部变量表、帧数据和操作数栈，现在已经知道了局部变量表与帧数据的分配机制及内存组成，只剩下操作数栈。不过操作数栈相比于局部变量表，却是更加变化不定，并且与被调用的 Java 方法息息相关，紧密相邻。这里先总结下 Java 方法栈帧中固定的部分，如图 7.28 所示。

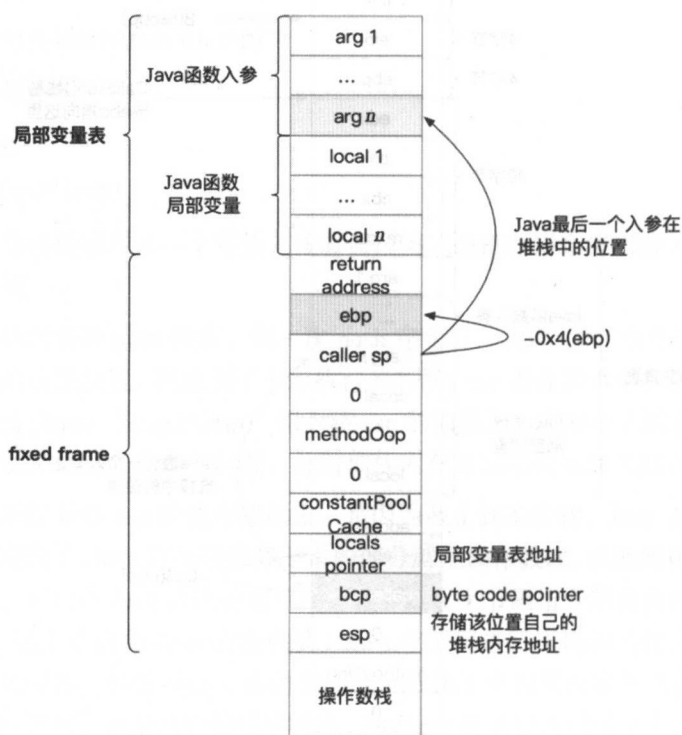


图 7.26 Java 栈帧的详细结构

Java 栈帧与 C/C++ 的栈帧自然是大不相同，Java 栈错综复杂，并且结构也是怪异到了极点，不像 C/C++ 等编程语言的栈帧那般简单明了。这些怪异点主要体现在下面几点。

1) 栈底与局部变量表底部不是同一位置

Java 方法的局部变量表包含两部分，分别是 Java 方法入参区域和 Java 内的局部变量区域。但是 Java 方法入参区域并不属于 Java 方法的栈帧，而是属于其调用方的栈帧的一部分。因此，

Java 方法栈由局部变量表、操作数栈和帧数据 (fixed frame) 组成, 这一说法并不够严谨, 因为 Java 方法仅包含了局部变量表的一部分区域。

由于 Java 方法的人参区域与内部局部变量区域分属于两个不同的方法, 因此当被调用的函数执行完成, 需要回收其堆栈空间时, 必须标记其真实的栈底位置, 因此 Hotspot 只能在 fixed frame 中保存 Java 方法的最后一个人参的堆栈内存位置, 这个位置恰好就是调用方的栈顶, 也是被调用的 Java 方法的栈底位置。HotSpot 通过这个位置来标记所需要回收的堆栈内存范围。

虽然 Java 方法的人参区域与内部局部变量区域分属于两个不同的方法, 但是对于被调用的一方而言, 这两个区域又必须在物理上连成一片, 这样才能形成一个整体, 作为一个完整的局部变量表, 方便被调用的 Java 方法的字节码指令进行读写, 因此 Hotspot 只能将调用方的栈底指针 bp 保存到 fixed frame 之中。这虽然未尝不可, 但是毕竟与 C/C++ 之类的编程语言有很大差异, 在程序逻辑上相差甚远。

2) Java 栈帧需要额外保存若干数据

Java 的 fixed frame 部分的数据其实与 Java 的源程序指令没有丝毫关系, 但是 fixed frame 中却保存了大量的运行时数据, 这一点也与其他编程语言差别较大。例如 C/C++, 编译后的栈帧几乎全部用于保存函数的人参和局部变量, 除了寥寥几个寄存器上下文的现场保存, 几乎就没有其他额外的数据。Java 栈帧的这种特性所带来的结果就是运行同样的逻辑, 其堆栈空间需要占用更多的内存。

这也难怪, 毕竟 JVM 仅仅只是使用软件模拟的虚拟系统而已, 由于硬件资源有限, 因此只能使用软件的方式来保存上下文了。例如 Java 方法的 fixed frame 中需要保存 bcp, 而在 C/C++ 等编程语言中不需要保存这么一个指向指令的指针, 自有那硬件寄存器 (ip 段寄存器) 自动存储。再如 Java 栈帧中的 caller sp, 在 C/C++ 语言中更加不需要使用宝贵的堆栈内存去保存, 也有那硬件寄存器去存储 (bp)。

当然, 虽然 Java 方法栈帧的结构十分怪异, 但是却自有其道理, 这由 JVM 自身的执行引擎和内存模型所决定。佛曰: 种善因, 得善果; 种恶因, 得恶果。同样, JVM 为其天生所秉持的崇高理想而设计的各种巧妙机制, 最终决定 Java 的方法栈只能长成这个样子, 多一分不行, 少一分更不行。

11. 创建 fixed frame 的全部脚本

Hotspot 调用 TemplateInterpreterGenerator::generate_fixed_frame(bool native_call) 函数创建 fixed frame, 这个函数与其他例程一样, 在 JVM 启动之初就会全部转换为机器指令, Hotspot 只需直接调用机器指令即可完成 Java 帧数据的创建。这部分所对应的机器指令 (32 位 x86 平台) 如下:

```

0xb36d6561: push    %eax          --恢复 return from java
0xb36d6562: push    %ebp          --enter
0xb36d6563: mov     %esp,%ebp
0xb36d6565: push    %esi          --最后一个参数在堆栈中的位置
0xb36d6566: push    $0x0
0xb36d656b: mov     0x8(%ebx),%esi --ebx 指向 method, 0x8(%ebx) 指向
                        ConstMethod
0xb36d656e: lea     0x30(%esi),%esi --在这一步让%esi 指向 codebase, 也就是
                        method 的第一个字节码的位置
0xb36d6571: push    %ebx          --将 method 入栈
0xb36d6572: mov     0x10(%ebx),%edx --method 性能数据
0xb36d6575: test    %edx,%edx
0xb36d6577: je      0xb36d6580
0xb36d657d: add     $0x58,%edx
0xb36d6580: push    %edx          --method 性能数据
0xb36d6581: mov     0xc(%ebx),%edx --ConstMethod *
0xb36d6584: mov     0xc(%edx),%edx --ConstantPoolCache *
0xb36d6587: push    %edx
0xb36d6588: push    %edi          --argument word 1 所在堆栈位置 (即局部变量
                        表第 1 个槽位位置)
0xb36d6589: push    %esi          --method 第一个字节码位置入栈
0xb36d658a: push    $0x0          --操作数栈栈底保留字段
0xb36d658f: mov     %esp, (%esp)  --将当前栈顶地址保存到操作数栈栈底 (帧数据栈
                        顶就是操作数栈栈底)

```

当 `TemplateInterpreterGenerator::generate_fixed_frame(bool native_call)` 函数执行完之后 (是指其所生成的机器指令执行完成, 该函数本身会在 JVM 启动之初被调用执行, 但是只会生成对应的机器指令, 并不会直接执行), 物理寄存器又有一番变化, 最终各个寄存器中所存储的结果如表 7.4 所示。

表 7.4 物理寄存器当前状态

寄存器名	指 向
edx	constantPoolCache
ecx	Java 函数入参数量
ebx	指向 Java 函数, 即 Java 函数所对应的 methodOop 对象
esp	Java 方法栈的 fixed frame 顶部

续表

寄存器名	指 向
ebp	Java 方法调用方的栈底
esi	bcp
edi	locals pointer
eax	return address

其中,尤其要关注 esi 与 edi 这两个寄存器,接下来 HotSpot 执行 Java 字节码指令时,字节码对 Java 方法的局部变量表的读写主要就依靠 edi 来进行相对寻址,而 Hotspot 调用字节码指令的前提是得先定位到首个字节码指令在内存中的位置,这个位置就存储在 esi 寄存器中。

7.4.3 局部变量表

前面详细讲解了局部变量表的创建过程及组成,其实说白了,局部变量表就是 JVM 为 Java 方法内的变量在堆栈上所分配的一块连续的内存空间而已,这块连续的内存空间用于存储 Java 方法的入参数据和局部变量。

1. 局部变量表的基本单位

在 JVM 内部,局部变量表按照“槽位”(slot)为基本单位进行划分。那么一个槽位 slot 的单位是多大呢?JVM 规范中给出了槽位单位的长度。

JVM 规范规定一个 slot 槽位应该能存放一个 boolean、byte、char、short、int、float、reference 或 returnAddress 类型的数据,其中 reference 是对象的引用,可以查到对象在 Java 堆中的实例的起始地址索引和方法区中的对象数据类型。returnAddress 是为字节码指令 jsr、jsr_w 和 ret 服务的,它指向了一条字节码指令的地址,其实说白了就是一个指针类型的数据。

但是 JVM 的这一对于 slot 长度的规范让人十分迷惑,因为在 JVM 内部,一个 boolean 类型与一个 float 类型所占的内存空间大小肯定是不同的,而 JVM 规范则规定一个 slot 槽位能够存放这两种数据类型的任何一种数据,那么 slot 到底是多大?

这是一个比较复杂的问题,各位道友继续往下看。

2. 局部变量表的读写与线程安全性

Java 程序员无法直接编写 Java 代码对局部变量表进行读写,毕竟这不是 Java 这种语言级别的数据结构概念,这种结构只能通过字节码指令进行访问和写入,并由 JVM 在运行期进行动态

读写。事实上，局部变量表作为堆栈的一部分，其实也决定了其只能由机器指令或者 JVM 这种能够创建和销毁堆栈的虚拟机器访问。

字节码指令也不能由 Java 程序员去编写，它由编译器生成。当涉及对局部变量表的访问时，编译器会生成 load 和 store 这样的指令，分别对局部变量表进行读和写。

例如下面这个简单的例子：

清单：Calculator.java

作用：计算器程序

```
class Calculator{
    public static int add(int x, int y){
        int sum = x + y;
        return sum;
    }
}
```

编译这个类，并使用 javap 命令分析编译后的字节码文件，得到该示例中的 add(int, int)方法的字节码指令是：

```
public static int add(int, int);
Code:
    Stack=2, Locals=3, Args_size=2
    0:   iload_0
    1:   iload_1
    2:   iadd
    3:   istore_2
    4:   iload_2
    5:   ireturn
```

对于 add()方法中 int sum = x + y 这样的源代码，既涉及对两个入参 x 和 y 的读取，也涉及对 sum 局部变量的写入，编译器将其翻译成 iload_0、iload_1、iadd 和 istore_2 这 4 条字节码指令。由于变量 x 和 y 是 add()方法的入参，因此其 slot 索引号分别是 1 和 2，所以 iload_0 和 iload_1 分别表示从局部变量表中读取变量 x 和变量 y。

当执行 iadd 指令完成 x 和 y 的求和运算之后，求和结果被存储到变量 sum 中。由于 sum 是 add()方法中的第 3 个局部变量，因此其在局部变量表的 slot 的索引号为 2，所以执行完 iadd 指令后，便通过 istore_2 指令将求和结果存储到 sum 变量中。

局部变量表的大小（或谓深度）由编译器直接在编译期间计算出来，因此 Java 方法的入参和局部变量的 slot 索引号在编译期便确定下来，所谓 slot 索引号，说白了其实就是变量以方法栈帧中的某个点作为基准位置进行偏移，终究是堆栈空间的一部分，运行时无法修改，在这一点上，Java 语言与其他众多语言都保持一致，虽然其他语言中未必就有局部变量表的概念，但

是在堆栈上一定为其分配了空间。不过在 C/C++ 语言中, 可以通过嵌入汇编脚本或者直接调用机器指令而动态扩展/收缩栈帧空间, 改变栈顶和栈底位置, “世界尽在掌握中”。所涉虽远, 然而 JVM 却是个中高手, 在执行引擎中到处可见这类逻辑, 如前文所讲执行引擎中的 CallStub 等例程都使用这种逻辑。

当在运行期完成上述 `int sum = x + y` 这段逻辑的字节码指令之后, 会执行最后一条指令 `ireturn`, 结束该方法的调用。`ireturn` 指令所对应的机器指令会将 JVM 为该方法所申请的栈帧空间销毁掉, 由于局部变量表就包含在栈帧之中, 因此栈帧都被销毁之后, 局部变量表自然也被销毁, 至此, 局部变量表便完成其使命, 轻轻地来, 轻轻地走, 不带走一片云彩, 亦不留下丝毫痕迹。

至于 `iload` 和 `istore` 系列的字节码指令究竟如何完成局部变量表的读和写, 则会在下文继续深入探讨, 这里先熟悉下即可。

总体而言, 局部变量表的生命周期包括以下几个环节:

(1) 在编译期间, 编译器通过文法、语义解析, 计算出一个 Java 方法所需的局部变量表大小, 并写入 Java class 字节码文件的方法属性的 `code` 属性表中

(2) 在 JVM 加载 Java 类的时候, 会解析 Java class 字节码文件中的方法信息, 并解析出局部变量表的大小, 如此便将局部变量表的大小这个数据从文件系统加载到内存中。

(3) 当 JVM 准备调用 Java 方法时, 会为该方法创建栈帧, 而栈帧中就包含局部变量表所需的空间。局部变量表的创建过程已在上文详细讲解过。这一步局部变量表终于横空出世, Java 方法的人参和内部的局部变量们终于有了“安身立命”的场所, 终于有了一个“家”, 并且一人一个, 大家按照先来后到的顺序按序分配, 谁也别抢, 谁也别争。为了防止走错了家门, 每个地址都被加了门牌号, 这个门牌号就叫作“slot 索引”。真是一个“完美世界”。

(4) 当 JVM 具体执行 Java 方法时, 便调用 Java 的 `iload` 和 `istore` 系列指令对栈帧中的局部变量表的空间进行不断读取和写入。由于有门牌号, 因此大家并不会串错了门, 一切都是井然有序。

(5) 当 JVM 执行完 Java 方法后, Java 方法的栈帧空间会被销毁, 由于局部变量表就包含在栈帧空间内部, 因此连同局部变量表一起被销毁。

由于局部变量表建立在堆栈空间上, 因此是线程私有数据, 所以 JVM 对其所进行的一切操作都不用考虑并发安全问题。

3. slot 大小

虚拟机通过门牌号——slot 索引号来统一区分局部变量表, Java 方法里的人参和局部变量

一人一个门牌号。但是人有胖瘦，局部变量有大小，这个大小由数据类型所决定。

JVM 规范规定一个 slot 槽位应该能存放一个 boolean、byte、char、short、int、float、reference 或 returnAddress 类型的数据。但是这些不同类型的数据所需的内存空间是不同的，而 slot 的基本单位是确定不变的，JVM 是如何保证这点的呢？换言之，slot 的基本单位到底是多大，或者说一个槽位所占的内存空间到底是多大？

除了这一疑问，还有另一个更大的疑问，JVM 规范规定一个 slot 槽位能够存放一个 reference，但是同时又说如果是 long 型则需要 2 个 slot 来存放，问题是，在 64 位平台上，如果不开启指针压缩功能，则一个引用类型的变量所占的内存空间与 long 类型的变量应该是一样大小，都是 64 位，但是 JVM 的这一规范着实让人十分不解，一个 slot 对两种数据宽度完全一样的数据的要求不同，那么这个 slot 的基本长度究竟是多大？到底是按照 long 的标准还是按照 reference 这个引用类型的标准？

这个答案不知道，不过可以通过编写示例程序进行测试，因为 Java 类在编译时便能确定其局部变量表的大小，并能确定各个变量的索引号，通过索引号便能知道每一种变量类型究竟占多大空间了。示例程序如下：

清单：Calculator.java

作用：计算器程序

```
class Calculator{
    public long add(long x, long y){
        long z = x + y;
        return z;
    }
}
```

本示例程序的 add() 是类成员方法，因此在编译期实际上会生成 3 个人参，其中第一个人参是隐式的 this 指针引用。add() 方法包含两个人参，其类型都是 long，同时 add() 方法内部的变量也是 long 类型。由于 add() 的第一个隐式入参 this 指针是一个引用类型 reference，而 add() 方法内部的局部变量都是 long 类型，因此通过 add() 方法所对应的字节码指令就能知道引用类型和 long 类型的变量究竟占多大空间。

编译该类，使用 javap 命令查看字节码，如下：

```
public long add(long, long);
Code:
    Stack=4, Locals=7, Args_size=3
    0:   lload_1
    1:   lload_3
    2:   ladd
    3:   lstore 5
```

```

5:    lload    5
7:    lreturn

```

javap 命令作的输出结果显示, add()方法的 locals=7, 这表示 add()方法的局部变量表包含 7 个 slot 槽位, 这很好理解, 由于 add()方法包含 1 个 this 引用类型的入参和 3 个 long 类型的局部变量, 一个引用入参占 1 个槽位, 而每个 long 类型的变量占 2 个槽位, 因此一共需要 7 个槽位。同时 javap 的输出显示 add()方法的入参 args_size=3, 表示 add()方法包含 3 个入参, 这是因为编译器会将 this 指针当作第一个入参。从 locals 和 args_size 这两个输出可以知道, JVM 内部的确让引用类型只占 1 个 slot, 而让 long 类型的数据占用 2 个 slot, 这与 JVM 的规范是一致的。

接着观察 add()方法的指令进行进一步确认。第 1 条字节码指令是 lload_1, 这表示读取 add()方法的第 1 个入参 x, 其实此时 add()方法的第 0 个入参是隐式的 this 指针, this 指针的 slot 索引是 0。而第一个入参 x 的 slot 索引为 1, 这的确表示 this 指针仅占用 1 个 slot。add()方法的第 2 条字节码指令是 lload_3, 该指令读取 add()方法的第 2 个入参 y, 注意此时 lload 后面的操作数变成 3, 表明第 1 个入参 x 占用了 2 个 slot, 否则如果只占用 1 个 slot, 则入参 y 的读取指令应该为 lload_2。

剩下的字节码指令各位道友自行推敲。总之无论从 locals 与 args_size 这些参数值, 还是从字节码指令看, 验证结果都与 JVM 的规范是完全一致的。不过这更加深了疑惑, 在 64 位平台上, 引用类型 reference 的变量所占的内存空间应该与 long 类型的数据是一致的, 但是为什么它们所占的 slot 槽数不同, 这里面究竟有何秘密?

JVM 规范并没有给出答案, 不过可以换个思路进行验证。JVM 将局部变量表的每个 slot 都编上了门牌号, Java 方法的每个入参和每个局部变量都有一个唯一的门牌号, JVM 在运行期会将这个“门牌号”转换为机器指令中内存数据传送时的偏移量, 可以通过偏移量观察到每个数据类型所占据的真实的内存空间大小。JVM 在创建 Java 方法栈帧时, 将局部变量表的起始位置保存到 edi 寄存器中, 并且局部变量表的所谓起始位置其实是 Java 方法第一个入参的内存位置, 这在前文有所提及。JVM 准备调用一个 Java 方法之前, 会将 Java 方法的 N 个入参进行压栈, 复制到局部变量表中, 但是此时 JVM 并未关注这些入参的实际类型, 而是将其统一当成指针类型进行处理, 因此在 32 位平台上, 这第一个入参在堆栈的内存位置, 其实是按照其数据宽度为 4 字节进行计算的, 这里就存在一个问题: 如果 Java 方法的第一个入参类型是 long, 则最终存储到 edi 寄存器中的所谓第一个入参的内存位置, 便不能代表真实的第一个入参的内存位置。如下面这个例子:

清单: Calculator.java

作用: 计算器程序

```

class Calculator{
    public static long add(long x, long y, long z){

```

```

    long sum = x + y;
    return sum;
}

```

该示例中的 `add(long, long, long)` 是个静态方法，因此没有隐式的 `this` 入参，其第一个入参就是声明中的 `long x`。JVM 准备调用该方法之前，会初始化该方法的局部变量表，初始化的总体思路在前文已经详细描述过，总体上分为两步（32 位平台）：

（1）先申请 N 个指针宽度的堆栈空间， N 为 Java 方法入参数量。对于本例而言，由于包含 3 个入参，同时在 32 位平台上，因此 JVM 先分配 12 字节内存空间。

（2）接着申请 $(\text{maxLocals} - \text{sizeOfParameters})$ 个指针宽度的堆栈空间。对于本例而言，使用 `javap` 命令查看编译后的字节码文件可知，其 `maxLocals=8`，`sizeOfParameters=3`，因此 JVM 会接着申请 5 个指针宽度的内存空间，在 32 位平台上，该内存空间一共包含 20 字节。

在第一步中，JVM 会顺便计算第一个入参的内存位置，并将其保存到 `edi` 寄存器中。但是这里所谓的第一个入参的内存位置，并非是该入参的真正的内存位置，因为此时仅仅是先申请堆栈空间并全部初始化为 0，尚未运行 Java 字节码指令将真正的局部变量存储进来。此时所谓第一个入参的位置，其实是 JVM 将各个入参当作指针看待时的位置。这里是一个关键点。

当上面这两步都执行完之后，JVM 便为 `add(long, long)` 方法分配好局部变量表的空间，其内存布局如图 7.27 所示。

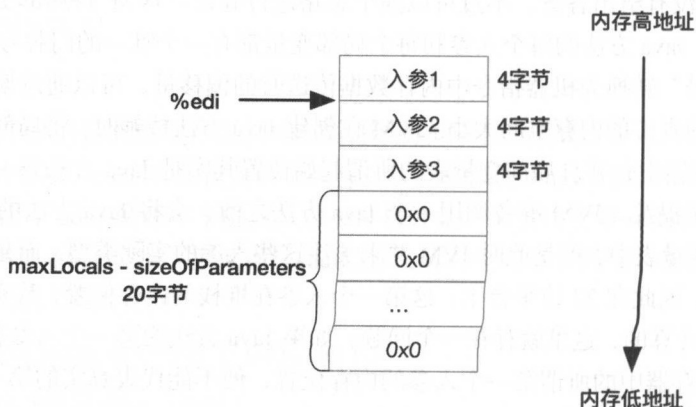


图 7.27 `add()` 方法的局部变量表初始化时的内存布局及 `edi` 寄存器指向位置

但是 `add(long, long, long)` 方法的第一个入参类型其实是 `long`，`long` 类型的数据需要占据 8 字节内存空间，因此该入参的真实内存起始位置应该是图 7.28 所示入参 2 所在的内存位置。

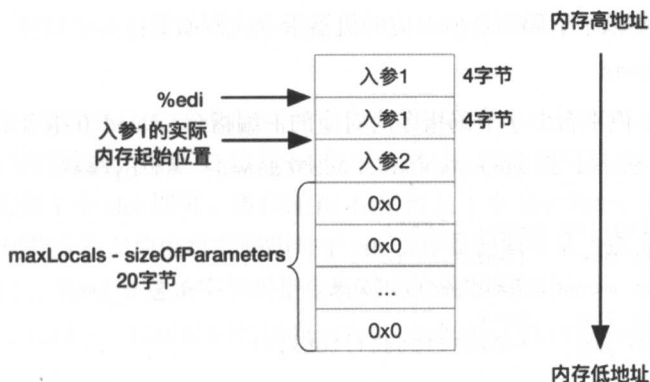


图 7.28 add()方法第一个入参的实际内存位置

图 7.28 给出一个很明显的信息：`edi` 寄存器所存储的位置，并非是局部变量表中第一个变量的实际内存位置。这种特性直接影响到 Java 字节码指令中读写局部变量表的 `load` 和 `store` 系列指令所对应的机器码层面的处理，因为 `load` 和 `store` 系列指令所对应的机器码在读写局部变量表时，实际上将 `edi` 寄存器所指的位置当作基准偏移位置，而现在 `add(long, long, long)` 方法的第一个 `long` 类型的入参的实际内存位置与 `edi` 寄存器这个基准位置不同，最终反映到机器码层面，必定要基于 `edi` 进行偏移。读取局部变量表中第一个 `long` 类型的数据的字节码指令是 `lload_0`，且看该指令在 32 位平台上所生成的机器指令：

```
lload_0 30 lload_0 [0xb36d8820, 0xb36d8860] 64 bytes
```

```
[Disassembling for mach='i386']
0xb36d8844: mov    -0x4(%edi),%eax
0xb36d8847: mov    (%edi),%edx
0xb36d8849: movzbl 0x1(%esi),%ebx
0xb36d884d: inc    %esi
0xb36d884e: jmp    *-0x48f102a0(,%ebx,4)
```

注意，第一条机器码指令是 “`mov -0x4(%edi), %eax`”，机器码果然从 `edi` 寄存器所指的下一个 4 字节的位置开始读取局部变量表中的 `long` 类型的数据，并将其保存到 `eax` 寄存器中。由于在 32 位平台上，一次 `mov` 指令最大只能传送 4 字节数据，因此这里对于 `long` 类型的数据连续使用了 2 条 `mov` 指令，连续读取 2 个 4 字节数据并分别保存到 `eax` 与 `ebx` 寄存器中。这里其实是使用了栈顶缓存技术，这里暂且不表，总之由此可以验证，`edi` 寄存器中所存储的的确不是第一个入参的真实内存起始位置。对于这一点，还有一个验证点，如果 Java 方法的第一个入参类型是 `int`，在 32 位平台上，Java 中的一个 `int` 类型数据占 4 字节内存空间，这与指针类型的数据宽度是一致的。那么如果 Java 方法的第一个入参若果真是 `int` 类型，则 `edi` 寄存器所指向的位置便与该入参的真实内存位置相同。若 Java 中读取局部变量表第一个数据且类型是 `int` 的字节

码指令是 `iload_0`，则该字节码指令所对应的机器指令应该如下：

```
mov (%edi), %eax
```

查看 JVM 在 32 位平台上字节码指令所对应的汇编指令，`iload_0` 指令如下：

```
iload_0 26 iload_0 [0xb36d86a0, 0xb36d86e0] 64 bytes
```

```
[Disassembling for mach='i386']
0xb36d86c4: mov    (%edi), %eax
0xb36d86c6: movzbl 0x1(%esi), %ebx
0xb36d86ca: inc    %esi
0xb36d86cb: jmp    *-0x48f106a0(, %ebx, 4)
```

结果果然与预测中的完全一致！所以如果 Java 方法的第一个入参类型是 `int` 类型，则 `edi` 寄存器所指的内存位置便与第一个入参的真实内存位置保持一致。

搞清楚了这一层的原理（前文说这是巨坑，道理就在这里，大脑得学会急转弯，否则就要撞树了），接着便可以验证 JVM 规范中所说的一个 slot 究竟占多大内存空间的问题了。

在 32 位平台上，一个 slot 能够存放一个 `int` 类型，也能存放一个 `reference` 类型，因此如果一个 Java 方法的第 1 和第 22 个入参的类型都是 `int` 类型，则读取局部变量表中这 2 个入参的字节码指令分别是 `iload_0` 和 `iload_1`，如果要验证 32 位平台上一个 slot 槽位究竟占多大内存空间，只需要分析 `iload_0` 与 `iload_1` 这 2 条字节码指令所对应的机器指令中，分别相对于 `edi` 寄存器做多大的偏移量即可。刚刚展示了 `iload_0` 字节码指令所对应的机器指令，那么在 32 位平台上，`iload_1` 字节码指令所对应的机器指令如下：

```
iload_1 27 iload_1 [0xb36d8700, 0xb36d8740] 64 bytes
```

```
[Disassembling for mach='i386']
0xb36d8724: mov    -0x4(%edi), %eax
0xb36d8727: movzbl 0x1(%esi), %ebx
0xb36d872b: inc    %esi
0xb36d872c: jmp    *-0x48f106a0(, %ebx, 4)
```

可以看到，最终机器指令从局部变量表中读取第二个且数据类型是 `int` 类型的变量时，使用了“`mov -0x4(%edi), %eax`”这条机器指令，这表示第二个 `int` 类型的变量在局部变量表中的起始位置相对于 `edi` 所指的位置偏移了 `-0x4` 字节，而这个偏移距离正好是 32 位平台上一个 slot 槽位的大小，由此可以验证 32 位平台上一个 slot 槽位占 4 字节内存空间。由于在 32 位平台上，一个指针类型的数据宽度也是 4 字节，而 Java 内部的 `reference` 引用类型的数据其实就是一个指针，因此 32 位平台上的 1 个 slot 槽位也能容纳下一个 `reference` 类型的数据。

按照这种思路,可以顺便验证下 32 位平台上一个 long 类型的数据是否的确需要两个 slot 槽位。

按照同样的思路,验证最让人疑惑的 64 位平台上的 slot 大小。由于 JVM 规范规定一个 slot 能够容纳一个 int 类型的数据,并无 32 位与 64 位平台之分,因此即使在 64 位平台上,一个 int 类型的数据仍然只需要 1 个 slot 即可。要验证 64 位平台上 1 个 slot 大小,只需要看 `iload_0` 与 `iload_1` 这 2 条字节码指令所对应的机器码相对于 `edi` 寄存器的偏移量,这个偏移量便是 slot 的大小。在 64 位平台上, `iload_0` 这条字节码指令所对应的机器码如下:

```
iload_0 26 iload_0 [0x000000010439f7e0, 0x000000010439f840] 96 bytes

0x000000010439f810: mov    (%edi),%eax
0x000000010439f813: movzbl 0x1(%esi),%ebx
0x000000010439f818: inc    %esi
0x000000010439f81b: jmp    *-0x103ceb100(,%ebx,8)
```

由第一条机器码可知, `iload_0` 指令读取局部变量表第一个且类型是 int 型的变量时,相对于 `edi` 寄存器的偏移量是 0。

64 位平台上, `iload_1` 字节码指令所对应的机器码如下:

```
iload_1 27 iload_1 [0x000000010439f860, 0x000000010439f8c0] 96 bytes

0x000000010439f890: mov    -0x8(%edi),%eax
0x000000010439f894: movzbl 0x1(%esi),%ebx
0x000000010439f899: inc    %esi
0x000000010439f89c: jmp    *-0x103ceb100(,%ebx,8)
```

由第一条机器码可知, `iload_0` 指令读取局部变量表第一个且类型是 int 型的变量时,相对于 `edi` 寄存器的偏移量是 -0x8。而上面 `iload_0` 指令相对于 `edi` 的偏移量是 0,由此可知 `iload_0` 与 `iload_1` 这 2 条字节码指令所读取的数据的内存位置的相对偏移量为 8 字节,而这正是 1 个 slot 的大小。

在进一步验证 64 位平台上的 slot 大小之前,有必要弄清楚在 64 位平台上 `edi` 寄存器所指的位置。当 JVM 准备执行一个 Java 方法时,先为其创建局部变量表并初始化为 0。还是以上面的 `Calculator.add(long x, long y, long z)` 方法为例,该方法的局部变量表创建完成之后,其内存布局及 `edi` 寄存器所指位置如图 7.29 所示。

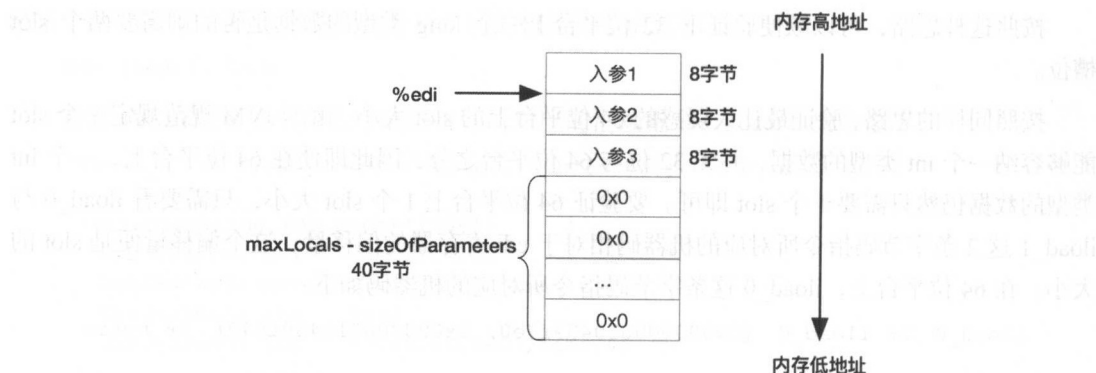


图 7.29 64 位平台上，add()方法的局部变量表初始化时的内存布局及 edi 寄存器指向位置

64 位平台与 32 位平台最大之不同便在于，在 JVM 创建局部变量表时，JVM 仍然将每一个入参当作一个指针类型的数据处理，但是一个指针在 64 位平台上占 8 字节，因此 edi 寄存器所指的位置，其后面其实能够容纳一个 long 类型的数据（即入参 1 的大小）。那么如果 Java 方法的第一个入参类型是 long，则 lload_0 字节码指令所对应的机器码应该直接从 edi 寄存器所指的位置处开始读取数据，那么来看下 64 位平台上，lload_0 这条字节码指令所对应的机器码，如下所示：

```
lload_0 30 lload_0 [0x0000000010439f9e0, 0x0000000010439fa40] 96 bytes
```

```
0x0000000010439fa10: mov    -0x8(%edi),%rax
```

```
0x0000000010439fa14: movzbl 0x1(%esi),%ebx
```

```
0x0000000010439fa19: inc    %esi
```

```
0x0000000010439fa1c: jmp    *-0x103ceb100(,%ebx,8)
```

关注第一条机器指令“mov -0x8(%edi), %rax”，这个结果令人吃惊，与预测的完全不同，lload_0 竟然不是从 edi 寄存器所指的位置开始读取 long 类型的数据，而是以 edi 所指位置为基准又向低地址方向偏移了 8 字节。由此可见，在 64 位平台上，一个 long 类型的数据在局部变量表中似乎占据了 16 字节的大小。不过这与 JVM 规范倒是完全相符，因为刚才验证过在 64 位平台上，一个 slot 的宽度为 8 字节，而 JVM 规范又规定存储一个 long 类型的数据需要 2 个 slot，而 2 个 slot 的宽度就是 16 字节。由此，谜底终于揭开了，JVM 的规范一点没错，无论是在 32 位平台还是在 64 位平台上都会一样生效。同时，这也解释了 64 位平台上的 reference 引用类型数据所占的 slot 数量为 1 的原因，因为虽然在 64 位平台上，一个引用类型本质上是一个指针类型，其所需内存大小按理应该与一个 long 类型的数据所需的内存大小相同，但是在 JVM 的局部变量表中，这两种类型的数据所需的 slot 数是不同的，这是由早期的 JVM 规范所规定的，所以到了 64 位平台上的 JVM，只能将一个 slot 实现为占据 8 字节大小的内存区域，如此才能让一个 slot 能容纳一个引用类型的数据。

不过在 JVM 内部, long 类型的变量也只有局部变量表中才会占据 16 字节,而在堆内存中,该类型的数据该占据多大的内存空间,还是占据多大的内存空间。例如,如果一个 Java 类的成员变量类型是 long 类型,则该变量会被分配在堆内存中,在堆内存中,该变量就只占 8 字节。从这个角度来看,64 位平台上的 JVM 的局部变量表对内存存在一定的空间浪费。

这里有个问题,可能有些道友认为既然可以使用 `iload_0` 与 `iload_1` 这两条字节码指令所对应的机器码相对于 `edi` 寄存器的偏移量之差来确定 slot 的大小,那么也应该能使用 `lload_0` 与 `lload_1` 这两条字节码指令来确定。例如,上面给出了 `lload_0` 字节码指令相对于 `edi` 的偏移量为 `-0x8`,而 `lload_1` 字节码指令在 64 位平台上所对应的机器码如下:

```
lload_1 31 lload_1 [0x000000010439fa60, 0x000000010439fac0] 96 bytes
```

```
0x000000010439fa90: mov    -0x10(%edi),%rax
0x000000010439fa94: movzbl 0x1(%esi),%ebx
0x000000010439fa99: inc    %esi
0x000000010439fa9c: jmp    *-0x103cebl00(,%ebx,8)
```

这条字节码指令从相对于 `edi` 偏移量为 `-0x10` 的位置开始读取局部变量。由于 `lload_0` 从相对于 `edi` 偏移量为 `-0x8` 的位置开始读取,这两个偏移量之差为 `0x8`,即 8 字节,由此得出 long 类型的数据在局部变量表中只需占据 8 字节大小的结论。很显然,这个结论是错误的,其根本原因在于,如果一个 Java 方法编译后能够产生 `lload_1` 这样的字节码指令,则说明该方法的局部变量表的第二个变量类型是 long,并且其索引号是 1,那么便说明 Java 方法的局部变量表中的第一个变量只占 1 个 slot 槽位,因此第一个变量类型一定是 int、reference 等类型,而绝不可能是 long 类型,否则第二个变量的读取指令就应该是 `lload_2`。如果第一个变量类型是 int,则其读取指令是 `iload_0`,因此需要比较 `iload_0` 与 `lload_1` 这两条字节码指令所读取的相对于 `edi` 寄存器所指内存的偏移量之差,由此才能确定局部变量表中一个 long 类型的数据所占据的 slot 槽数。很显然,`iload_0` 字节码指令从相对于 `edi` 偏移量为 0 的位置开始读取局部变量,而 `lload_1` 从相对于 `edi` 偏移量为 `-0x10` 的位置开始读取局部变量,这两者的偏移量之差为 `0x10`,即 16 字节,因此也能验证出一个 long 类型的数据在局部变量表中占据 16 字节的内存大小。这个结果与上述的验证结果完全一致。

至此,关于 JVM 内部局部变量表中的 slot 到底是多大的问题便验证完毕。总结如下:

- ◎ 32 位平台上,一个 slot 大小为 4 字节。
- ◎ 64 位平台上,一个 slot 大小为 8 字节。
- ◎ 64 位平台上, long 类型数据在局部变量表中占据 16 字节的内存空间,但是在堆内存中该占据多大内存空间还是占据多大。

不过根据 JVM 规范,还有一个疑问,那就是一个 slot 能够容纳一个 int、reference 等类型

的数据，还能容纳一个 short、char 等类型的数据。由于 char、short 所需内存空间，小于 int 与 reference 类型所需的内存空间，那么对于这类窄数据，JVM 如何在局部变量表中进行存取呢？看下面示例：

清单：Calculator.java

作用：计算器程序

```
class Calculator{
    public static int mixed(short s, char c){
        s = 32;
        c = 'D';
        short ss = (short)( s + (short)3);
        char cc = (char)(c + (char)2);
        int i = s + c;
        return i;
    }
}
```

编译程序，并使用 javap 命令分析编译后的字节码文件，javap 命令的分析结果输出如下：

```
public static int mixed(short, char);
```

Code:

```
Stack=2, Locals=5, Args_size=2
```

```
0:   bipush  32
2:   istore_0
3:   bipush  68
5:   istore_1
6:   iload_0
7:   iconst_3
8:   iadd
9:   i2s
10:  istore_2
11:  iload_1
12:  iconst_2
13:  iadd
14:  i2c
15:  istore_3
16:  iload_0
17:  iload_1
18:  iadd
19:  istore  4
21:  iload  4
23:  ireturn
```

从 javap 命令的分析结果可以看出,对于 char 和 short 类型的数据,无论是读还是写,所使用的指令全都是 iload 与 istore 系列的指令,因此在 JVM 内部,对于数据宽度小于 int 类型的数据类型,仍然将其处理成 int 类型的数据,并使用基于 int 类型数据的读写指令对 char、short 等类型的数据进行读写。有兴趣的道友可以自行设计程序验证 boolean 与 byte 类型的数据所对应的读写字节码指令是什么。

7.5 栈帧深度与 slot 复用

通常一个 Java 程序会包含多个线程,每个线程都会包含若干方法栈帧,这些若干方法栈帧就组成了线程的堆栈空间。线程堆栈空间 (stack space) 不会无限制地增长,而是会受到约束,直接由操作系统加载的软件程序的线程堆栈空间大小会受到操作系统层面所设置的栈空间大小限制,而对于 Java 线程,同时还会受到 JVM 虚拟机所设置的堆栈空间大小限制。

正是因为堆栈空间大小会受到限制,所以当在一个线程中所调用的方法太深时,导致 JVM 所分配的栈帧太多,就有可能耗尽 stack space,从而抛出 stackOverflow 异常。所以合理地设置默认堆栈空间大小是一门学问。在 JVM 中,可以通过 XSS 来设置默认的堆栈空间大小。

前面讲过,Java 方法栈帧由三大部分所组成:局部变量表、固定帧和操作数栈。固定帧的大小是固定不变的,无论所调用的是何种 Java 方法。因此 Java 方法栈帧的大小取决于局部变量表和操作数栈的大小,而由于调用者方法的操作数栈会作为被调用者方法的局部变量表的一部分(栈帧重叠),因此可以认为操作数栈属于被调用者方法的栈帧的一部分,由此看来,一个 Java 方法的栈帧大小主要取决于局部变量表的大小,而 Java 方法的局部变量表主要由两部分组成:一是 Java 方法入参;二是 Java 方法局部变量。

因此,Java 方法的栈帧大小最终取决于 Java 方法的入参和局部变量的大小。为何要分析这个问题呢?实在是因为这与线程堆栈的合理运用存在着莫大的关系。当 JVM 设定默认的堆栈空间大小后,一个 Java 线程所能调用的最大方法深度便直接取决于 Java 方法局部变量表的大小。如果 Java 方法的局部变量表所占的空间大,则 Java 线程所能调用的最大方法深度便会变小。

例如下面这个示例:

清单: TestXSS.java

作用: 演示 Java 堆栈空间大小与局部变量表的关系

```
public class TestXSS implements Runnable{
    public static void main(String[] args)
        Thread t = new Thread(new TestXSS());
        t.start();
}
```

```
}

public static void test(long a){
    long b = a + 3;

    System.out.println(a++);
    test(a);
}

@Override
public void run() {
    test(1);
}
}
```

该程序主要通过递归调用 `test(long)` 函数，来演示递归多少次之后会将线程堆栈空间耗尽。

设置 JVM 的 XSS 大小为 200KB，运行该程序，输出如下：

```
//...
950
951
952
953
Exception in thread "Thread-0" java.lang.StackOverflowError
    at sun.nio.cs.ext.DoubleByte$Encoder.encodeArrayLoop(DoubleByte.java:578)
    at sun.nio.cs.ext.DoubleByte$Encoder.encodeLoop(DoubleByte.java:617)
    at java.nio.charset.CharsetEncoder.encode(CharsetEncoder.java:579)
    at sun.nio.cs.StreamEncoder.implWrite(StreamEncoder.java:271)
    at sun.nio.cs.StreamEncoder.write(StreamEncoder.java:125)
    at java.io.OutputStreamWriter.write(OutputStreamWriter.java:207)
    // ...
```

可见，当递归调用 953 次之后，线程的堆栈空间终于被耗尽。当堆栈空间被耗尽后，JVM 便会抛出 `java.lang.StackOverflowError` 异常。

接着对上面的示例程序稍做修改，主要修改 `test(long)` 方法，修改后的方法如下：

清单：TestXSS.java

作用：演示 Java 堆栈空间大小与局部变量表的关系

```
public static void test(long a){
    long b = a + 3;
    long c = b - a + b * 5;
    long d = a & 6;

    System.out.println(a++);
    test(a);
}
```

修改后的 `test(long)` 方法内部另外声明了两个局部变量，并且都是 `long` 类型的。仍然将 JVM 的 `XSS` 参数设置为 200KB，运行修改后的程序，输出结果如下：

```
//...
739
740
741
742
743
Exception in thread "Thread-0" java.lang.StackOverflowError
  at sun.nio.cs.ext.DoubleByte$Encoder.encodeArrayLoop(DoubleByte.java:578)
  at sun.nio.cs.ext.DoubleByte$Encoder.encodeLoop(DoubleByte.java:617)
  at java.nio.charset.CharsetEncoder.encode(CharsetEncoder.java:579)
  at sun.nio.cs.StreamEncoder.implWrite(StreamEncoder.java:271)
  at sun.nio.cs.StreamEncoder.write(StreamEncoder.java:125)
  at java.io.OutputStreamWriter.write(OutputStreamWriter.java:207)
// ...
```

现在可以看到，`test(long)` 函数仅递归调用了 743 次便耗尽了线程的堆栈空间。由此可见，当 Java 方法的局部变量表增大后，的确会减少方法调用的深度。各位道友可以继续在此 `test(long)` 方法内部定义更多的局部变量并测试所能递归调用的最大次数。

由于局部变量表的大小直接影响到一个线程所能调用的方法深度，因此在声明方法局部变量时，应该尽量使 `slot` 能够复用。所谓 `slot` 复用，便是让方法内部的不同变量能够占用局部变量表中的同一个槽位，这样便能减小局部变量表的大小，从而提高一个线程能调用的最大方法深度。

仍然以上面修改后的 `test(long)` 方法作为示例，该方法内部包含 3 个局部变量，分别是 `b`、`c` 和 `d`，但是仔细观察可以发现，从源程序的 `d` 变量声明开始，一直到方法结束，`a` 和 `b` 变量都没有再被使用到，因此可以使用花括号 `{}` 将 `a` 和 `b` 变量的声明语句括起来，如下：

清单：TestXSS.java

作用：演示 Java 堆栈空间大小与局部变量表的关系

```
public static void test(long a){
    {
        long b = a + 3;
        long c = b - a + b * 5;
    }

    long d = a & 6;

    System.out.println(a++);
    test(a);
}
```

现在仍然将 JVM 的 XSS 设置为 200KB，运行该示例程序，输出结果如下：

```
// ...
831
832
833
834
835
836
Exception in thread "Thread-0" java.lang.StackOverflowError
    at sun.nio.cs.ext.DoubleByte$Encoder.encodeArrayLoop(DoubleByte.java:578)
    at sun.nio.cs.ext.DoubleByte$Encoder.encodeLoop(DoubleByte.java:617)
    at java.nio.charset.CharsetEncoder.encode(CharsetEncoder.java:579)
    at sun.nio.cs.StreamEncoder.implWrite(StreamEncoder.java:271)
    at sun.nio.cs.StreamEncoder.write(StreamEncoder.java:125)
    at java.io.OutputStreamWriter.write(OutputStreamWriter.java:207)
    at java.io.BufferedWriter.flushBuffer(BufferedWriter.java:129)
    at java.io.PrintStream.write(PrintStream.java:526)
    // ...
```

从输出结果可以看出，`test(long)`方法被递归调用的最大次数相比于刚才的 743 次增大到 836 次，由此可见增加花括号的的方式的确有效。这是因为使用花括号将前面两个变量括起来之后，Java 编译器便会认为其作用域不会超出花括号的范围，因此在出了花括号的范围之后，JVM 便会清空这两个变量所占用的 slot 槽位，空出来给 `test(long)`方法内部后续的变量复用。各位道友可以通过 `javap` 命令分别分析使用花括号前后，`test(long)`方法的 locals 值。

7.6 最大操作数栈与操作数栈复用

与 Java 方法堆栈息息相关的一个重要参数是 `max stack`，即最大操作数栈。Java 虚拟机的指令集基于栈，所有的计算逻辑都需要通过栈来完成，这个栈便是“操作数栈”，JVM 内部也叫“表达式栈”。操作数栈的大小由 Java 编译器在编译期计算，但是编译器只能计算出一个“最大”的栈深度，这是因为操作数栈与 slot 槽一样，也可以实现复用。而之所以要实现复用，是为了节省宝贵的内存空间。最大操作数栈被作为 Java class 字节码文件内部 `Code` 属性区的一部分，Java 源文件被编译后，使用 `javap -v` 命令可以查看每一个 Java 方法的最大操作数栈大小，例如下面这个例子：

清单：Test.java

作用：演示 Java 最大操作数栈

```
public class Test {
    public static void main(String[] args){
```



```

        int a = 0;
    }
}

```

使用 `javap -v` 命令分析编译后的 `class` 字节码文件，输出如下信息：

```

public static void main(java.lang.String[]);
Code:
    Stack=1, Locals=2, Args_size=1
    0:   iconst_1
    1:   istore_1
    2:   return

```

打印结果中的“`stack=1`”，表示本程序的 `main()` 主函数的最大操作数栈空间只需要 1 个即可。`stack` 的数据宽度与 `slot` 槽位宽度保持一致，因此这里也可以说，`main()` 主函数的最大操作数栈空间为 1 个槽位。在本示例程序中，`main()` 主函数的字节码指令中包含“`iconst_1`”，该指令会将自然数 1 推送至栈顶，因此 `main()` 主函数需要 1 个槽位大小的操作数栈空间。

修改上面的示例程序，变成如下：

清单：Test.java

作用：演示 Java 最大操作数栈

```

public class Test {
    public static void main(String[] args){
        int a = 1;
        int b = 1;
    }
}

```

现在在 `main()` 主函数中定义了 2 个变量 `a` 和 `b`，使用 `javap` 命令分析后得知 `main()` 主函数的最大操作数栈仍然是 1，这是因为当 JVM 执行 `main()` 函数中的“`int a=1`”这条指令时需要将自然数 1 推送至栈顶，但是当执行完之后，栈顶的自然数 1 会从栈顶被传送至局部变量表中，因此当执行“`int b=1`”时，JVM 便可以复用栈顶的 1 个空间。

接着看下面这个示例：

清单：Test.java

作用：演示 Java 最大操作数栈

```

public class Test {
    public static void main(String[] args){
        int a = 1;
        int b = 1;

        add(a, b);
    }
}

```

```

    public static void add(int m, int n){
        int a = m + n;
    }
}

```

本示例与上面的示例相比,在 main()函数里面多了一步——调用 add()方法。使用 javap 命令分析,输出结果如下:

```

public static void main(java.lang.String[]);
Code:
    Stack=2, Locals=3, Args_size=1
    0:  iconst_1
    1:  istore_1
    2:  iconst_1
    3:  istore_2
    4:  iload_1
    5:  iload_2
    6:  invokestatic    #2; //Method add:(II)V
    9:  return

```

可以看到,现在 Stack 的值为 2。为何会是 2 呢?这主要是因为 main()主函数调用了 add()方法。由于 add()方法包含 2 个入参,因此当 main()函数调用 add()方法时,需要将 add()方法所需的两个实参推送至 main()方法的操作数栈栈顶。JVM 在执行 Java 方法调用时,实现了“堆栈重叠”技术,因此这两个栈顶实参将被当作 add()方法局部变量表的一部分。在本例中,如果没有操作数栈复用技术,则 main()函数的最大操作数栈一定为 4,但是由于“int b=1”指令复用了“int a=1”指令的操作数栈空间,而“add(a,b)”指令又复用了“int b=1”指令的操作数栈空间,因此最终 main()函数只需要分配 2 个槽位大小的操作数栈空间,便能满足全部指令的逻辑计算。由此可以明白 Java 方法的操作数栈之“最大”的含义——这个“最大”实则是在内存复用的基础上,从一个 Java 方法所有指令中选出一个需要占用最多操作数栈空间的指令,以该指令所需的操作数栈空间作为一个 Java 方法的“最大”操作数栈空间。

明白了这个道理,各位道友可以猜一猜下面这个示例中 main()主函数的最大栈是多少:

清单: Test.java

作用: 演示 Java 最大操作数栈

```

public class Test {
    public static void main(String[] args){
        int a = 1;
        int b = 1;

        add(a, b);
        add(a, a, a);
    }
}

```

```

public static void add(int m, int n){
    int a = m + n;
}

public static void add(int m, int n, int x){
    int a = m + n + x;
}
}

```

7.7 本章总结

如果你能阅读到这里，说明你的功力非常深厚，或者你是一个十分有耐心的道友。耐得住寂寞，才能守得住繁华，有耐心的人必有所成就！

本章的难度在全书而言，属于难度系数最高的部分，一方面，涉及函数指针，另一方面，Java 方法栈帧的创建全依赖机器指令，这对于一般人而言，几乎等同于天书。在前面讲解 CallStub 例程的章节中，这两方面大家都见识过，然而，本章另一个难点在于栈帧的结构。

本章全面分析了 Java 方法栈帧创建的过程，机器指令几乎是逐个讲解的。然而，笔者在阅读 JVM 的这部分机制指令的过程中，不仅仅停留于分析机器指令本身的含义，还进行了更深入的思考，仔细推敲了每一条机器指令的背景，为何要这么实现，如果不这么实现有没有问题。相信真正有耐心读下来并且能够读懂的道友能够体会笔者的这种深入思考！

这也是本书区别于其他书籍的最大不同点，不仅追求“知其然”，更加追求“知其所以然”。

第 8 章

类方法解析

本章摘要

- ◎ Java 方法签名解析
- ◎ Java 方法的 code 属性解析
- ◎ LVT 与 LVTT
- ◎ method 创建
- ◎ Java 方法的字节码指令解析
- ◎ <clinit>()方法与<init>()方法
- ◎ 使用 HSDB 查看运行时的字节码指令
- ◎ vtable 的概念与机制

前面分析了 HotSpot 解析常量池和类变量的详细过程，这一章接着探讨类方法的解析。作为一门面向对象的语言，每一个 Java 类都有属性和行为这两个基本要素，而 Java 又作为一门解释性的语言，由 JVM 虚拟机负责解释执行。JVM 执行的正是 Java 类的“行为”——Java 方法，而执行之前，必须要先对方法进行解释，毕竟 JVM 虚拟机没有真正的运算能力，最终必须依靠物理 CPU 完成 Java 字节码的运算。

Java 方法的解析大体上可以分为 3 道工序：

(1) 在 Java 类源代码编译期间，编译器负责将 Java 类源代码翻译为对应的字节码指令，同时完成的工作还有 Java 方法局部变量表的计算，以及最大操作数栈的计算。

(2) 在 JVM 运行期间，JVM 加载类型，调用 `classFileParser::parseClassFile()` 函数对 Java class 字节码文件进行解析，在这一步将会完成 Java 方法的分析、字节码指令存放、父类与接口类方

法继承与重载等一系列逻辑。

(3) 在调用系统加载器 System Class Loader (SCL) 对应用程序的 Java 类进行加载的过程中, 完成方法符号链接、验证, 最重要的是完成 vtable 与 itable 的构建, 从而支持在 JVM 运行期的方法动态绑定 (也叫晚绑定)。当然, Java 技术体系不仅提供了 SCL, 还提供了其他类加载器用于加载 Java 类, 开发者也可以自定义类加载器来加载, 关于类加载器的话题会在本书后续章节详细讨论。

经过这 3 道层层推进的工序, 最终才能完成 Java 类方法的解析工作, 等这一切都完成后, JVM 才能在运行期通过 `invoke_virtual` 等字节码指令, 完成 Java 方法的调用和执行。

本章主要讲述 Java 类方法解析的第 2 道工序, 该工序主要在 `classFileParser::parseClassFile()` 函数中完成。`classFileParser::parseClassFile()` 函数主要完成 Java 类 class 字节码文件的解析, 其中不仅仅包含 Java 方法的解析, 还包含其他步骤, 前面几个步骤在前文都已进行了详细分析。这里还是按照惯例给出这个函数的总体“地图”(见图 8.1), 以理顺思路。

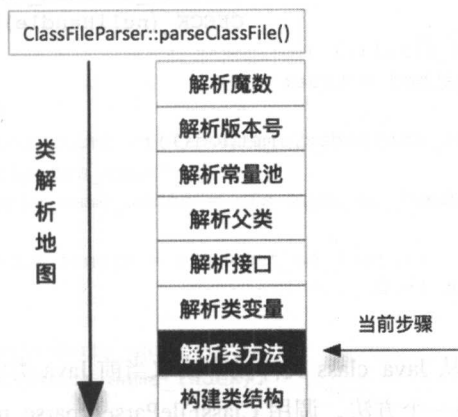


图 8.1 类解析的当前步骤——类方法解析

`classFileParser::parseClassFile()` 函数通过调用 `ClassFileParser::parse_methods()` 函数完成 Java 类方法解析的第 2 道工序。

`ClassFileParser::parse_methods()` 函数主要逻辑如下 (仅保留主要逻辑, 不相关的代码皆已去除, 以免浪费纸张):

清单: `/src/share/vm/classfile/classFileParser.cpp`

作用: `parse_methods()` 方法主要逻辑

```
objArrayHandle ClassFileParser::parse_methods(constantPoolHandle cp, bool
is_interface,
```

```

        AccessFlags* promoted_flags,
        ...) {
    ClassFileStream* cfs = stream();
    objArrayHandle nullHandle;
    typeArrayHandle method_annotations;
    typeArrayHandle method_parameter_annotations;
    typeArrayHandle method_default_annotations;
    u2 length = cfs->get_u2_fast();//获取方法长度
    if (length == 0) {
        return objArrayHandle(THREAD, Universe::the_empty_system_obj_array());
    } else {
        objArrayOop m = oopFactory::new_system_objArray(length, CHECK_(nullHandle));
        objArrayHandle methods(THREAD, m);
        for (int index = 0; index < length; index++) {
            methodHandle method = parse_method(cp, is_interface,
                                                promoted_flags,
                                                &method_annotations,
                                                &method_parameter_annotations,
                                                &method_default_annotations,
                                                CHECK_(nullHandle));

            if (method->is_final()) {
                *has_final_method = true;
            }
            methods->obj_at_put(index, method());
        }

        return methods;
    }
}

```

这段逻辑很简单，首先从 Java class 文件流中读取当前 Java 类中所定义的全部方法数量，接着循环遍历 Java 类中的每一个方法，调用 `ClassFileParser::parse_method()` 函数对 Java 方法进行逐个解析。

`ClassFileParser::parse_method()` 函数所包含的逻辑颇为复杂，毕竟这是 Java 类的核心所在。不过虽然复杂，却是条缕分明，层次清晰，总体看来，其源码“骨架”如下：

清单：/src/share/vm/classfile/classFileParser.cpp

作用：parse_method() 方法主要逻辑

```

methodHandle ClassFileParser::parse_method(constantPoolHandle cp, bool
is_interface,
                                                AccessFlags *promoted_flags,
                                                typeArrayHandle* method_annotations,
                                                typeArrayHandle*
method_parameter_annotations,

```

```

typeArrayHandle*
method_default_annotations,
    TRAPS) {
    ClassFileStream* cfs = stream();
    methodHandle nullHandle;
    ResourceMark rm(THREAD);

    // ①. 读取和验证 Java 方法的访问标识、名称
    int flags = cfs->get_u2_fast();
    u2 name_index = cfs->get_u2_fast();
    // ...

    // 定义方法相关的内部属性, 解析时会用到
    u2 max_stack = 0;
    u2 max_locals = 0;
    u4 code_length = 0;
    u1* code_start = 0;
    u2 exception_table_length = 0;
    // ...

    // ②. 解析方法的属性
    u2 method_attributes_count = cfs->get_u2_fast();
    while (method_attributes_count--) {
        u2 method_attribute_name_index = cfs->get_u2_fast(); // 获取 Java 方法当前
        属性的名称索引
        u4 method_attribute_length = cfs->get_u4_fast(); // 获取 Java 方法当前属
        性在字节码文件中所占的长度

        Symbol* method_attribute_name =
        cp->symbol_at(method_attribute_name_index);

        // Java 方法的 code 属性
        if (method_attribute_name == vmSymbols::tag_code()) {
            // ...
            while (code_attributes_count--) {
                // ...
                // 解析栈深度、局部变量表深度等

                // Java 方法源代码行号
                if (LoadLineNumberTables &&
                    cp->symbol_at(code_attribute_name_index) ==
                    vmSymbols::tag_line_number_table()) {
                    // Parse and compress line number table

```



```

        // ...

        // Java 方法局部变量表
        } else if (LoadLocalVariableTables &&
            cp->symbol_at(code_attribute_name_index) ==
vmSymbols::tag_local_variable_table()) {
            // Parse local variable table
            // ...

            // Java 方法局部变量类型表
            } else if (LoadLocalVariableTypeTables &&
                _major_version >= JAVA_1_5_VERSION &&
                cp->symbol_at(code_attribute_name_index) ==
vmSymbols::tag_local_variable_type_table()) {
                // ...
            }

            // ...
        }

        // 异常属性
        } else if (method_attribute_name == vmSymbols::tag_exceptions()) {
            // ...

            // 编译器生成属性--synthetic
        } else if (method_attribute_name == vmSymbols::tag_synthetic()) {
            // ...

            // 方法废弃属性
        } else if (method_attribute_name == vmSymbols::tag_deprecated()) { //
4276120
            if (method_attribute_length != 0) {
                classfile_parse_error(
                    "Invalid Deprecated method attribute length %u in class file %s",
                    method_attribute_length, CHECK_(nullHandle));
            }
        }
        // ...
    }

    // ③. 创建 methodOop
    methodOop m_oop = oopFactory::new_method(code_length, access_flags,
linenumber_table_length,
                                                total_lvt_length,
checked_exceptions_length,
                                                oopDesc::IsSafeConc,

```

```

CHECK_(nullHandle));
methodHandle m (THREAD, m_oop);

// ...

// ④. 复制字节码
m->set_code(code_start);

// ...

return m;
}

```

笔者使用①、②、③这样的标记将 `parse_method()` 方法分成了几个大的步骤，这几个步骤分别是：

- (1) 读取和验证 Java 方法的访问标识、名称。
- (2) 解析方法的属性。
- (3) 创建 `methodOop`。
- (4) 复制字节码。

本文将 HotSpot 解析 Java 方法的步骤分为这 4 大步，虽然 HotSpot 其实并不仅仅只做了这 4 件事，例如解析方法的注解等，只是这 4 个步骤是理解 Java 方法解析的精髓所在。

8.1 方法签名解析与校验

Java class 字节码文件中的方法属性部分的解析，从方法的 `flags` 标识的索引（指该标识在 Java class 内部常量池的索引号，下同）开始。`flags` 标识占用 2 字节长度，并且其后连续跟了 4 字节，分别是方法名称索引和方法描述索引，方法名称索引和描述索引各占 2 字节长度。Java 方法的签名信息主要由 3 部分组成：

- ◎ 方法的标识，`public`、`private`、`static`、`final`、`synchronized`、`native` 等
- ◎ 方法的名称
- ◎ 方法的描述，描述方法的返回值类型和入参信息，例如 `()V` 标识无入参的 `void` 类型方法

这 3 种信息共同组成 Java 方法的签名信息。由于每种信息中所存储的都是指向常量池的索引号，因此只需要 2 字节，Java 方法签名信息总共需要 6 字节长度。

在 `parse_method()` 方法一开始，连续调用 3 次 `cfs->get_u2_fast()`，分 3 次从 Java class 字节码文件中读取 Java 方法签名的 3 部分索引，并逐一进行校验。

在解析 Java 方法名称时，调用了 `classFileParser.cpp::verify_legal_method_name()` 方法校验 Java 方法名称的有效性，`verify_legal_method_name()` 主要逻辑如下：

清单：/src/share/vm/classfile/classFileParser.cpp

作用：verify_legal_method_name() 方法主要逻辑

```
void ClassFileParser::verify_legal_method_name(Symbol* name, TRAPS) {
    // ...
    char* bytes = name->as_utf8_flexible_buffer(THREAD, buf, fixed_buffer_size);
    unsigned int length = name->utf8_length();
    bool legal = false;

    if (length > 0) {
        if (bytes[0] == '<') {
            if (name == vmSymbols::object_initializer_name() || name ==
vmSymbols::class_initializer_name()) {
                legal = true;
            }
        } else if (_major_version < JAVA_1_5_VERSION) {
            // ...
        } else {
            legal = verify_unqualified_name(bytes, length, LegalMethod);
        }
    }

    // ...
}
```

在这段逻辑中，首先判断 Java 方法的第一个字符是否是“<”，如果是该字符，则接着判断当前 Java 方法是否是编译器自动生成的 `<init>` 和 `<clinit>` 这两种方法，如果不是，则校验不通过。所以，开发者所定义的 Java 方法，其名不能以“<”开始。

如果 Java 方法名不以“<”开始，则 `verify_legal_method_name()` 会调用 `verify_unqualified_name()` 函数继续校验 Java 方法中是否包含特殊的字符，`verify_unqualified_name()` 函数主要逻辑如下：

清单：/src/share/vm/classfile/classFileParser.cpp

作用：verify_unqualified_name() 方法主要逻辑

```
bool ClassFileParser::verify_unqualified_name(
    char* name, unsigned int length, int type) {
    jchar ch;

    for (char* p = name; p != name + length; ) {
        ch = *p;
```

```

if (ch < 128) {
    p++;
    if (ch == '.' || ch == ';' || ch == '[') {
        return false; // do not permit '.', ';', or '['
    }
    if (type != LegalClass && ch == '/') {
        return false; // do not permit '/' unless it's class name
    }
    if (type == LegalMethod && (ch == '<' || ch == '>')) {
        return false; // do not permit '<' or '>' in method names
    }
} else {
    char* tmp_p = UTF8::next(p, &ch);
    p = tmp_p;
}
return true;
}

```

这段逻辑表明, Java 方法名中不能包含如下特殊字符:

“.,;、[、/、<、>”

8.2 方法属性解析

在 `ClassFileParser::parse_method()` 函数中, 解析和校验完方法的签名信息, 接着就开始解析 Java 方法的属性。在前文详细分析 Java class 字节码文件的结构时讲到, 在字节码文件中使用几个大的属性来描述 Java 方法的方方面面的信息, 这些属性包括 `code`、`exception`、`line number table` 等。

Java 方法的几大属性并不一定每个都会在字节码文件中出现, 例如, 如果 Java 方法没有抛出异常, 则不会有异常表产生。同时, 字节码文件中描述属性的字节码区域之间并不是有序的, 因此, 字节码文件中仅存储 Java 方法的属性的总数量, 在 `ClassFileParser::parse_method()` 函数中根据属性的总数量, 通过 `while` 循环来逐个读取 Java 方法的当前属性, 通过属性名来判断当前究竟是哪个属性, 然后根据不同属性所承载的信息与结构之不同, 分别使用不同的策略进行解析。

8.2.1 code 属性解析

Java 方法的 `code` 属性解析都集中在 `ClassFileParser::parse_method()` 函数的 `while` 循环下的 `if (method_attribute_name == vmSymbols::tag_code())` 块中。

Java 方法的 code 属性主要包含属性的总长度、最大栈深度、局部变量表数量、字节码指令，除此以外，code 属性本身是一个复合属性，其下面还包含几个子属性，例如行号表、局部变量表等。

Java 方法的 code 属性起始于属性名称的常量池索引号，索引号之后所跟的是属性长度，所以在 `ClassFileParser::parse_method()` 中主要解析 Java 方法几大属性的 while 循环中，首先便是执行 `u2 method_attribute_name_index = cfs->get_u2_fast()` 和 `u4 method_attribute_length = cfs->get_u4_fast()` 来分别获取当前属性的索引号和总长度。Java 方法的几大属性的索引号的数据宽度为 2 字节，总数据宽度为 4 字节。

紧跟在 code 属性的总长度后面的是 3 个属性：`max_stack`、`max_locals` 和 `code_length`。所以在 `if(method_attribute_name == vmSymbols::tag_code()) {}` 块中连续调用 3 次 `get_u*_fast()` 来分别获取这 3 个属性数据。在不同的 JVM 版本中，`max_stack`、`max_locals` 和 `code_length` 所占用的数据宽度不同，HotSpot 对此进行了区分，逻辑如下：

清单：/src/share/vm/classfile/classFileParser.cpp

作用：parse_method() 处理不同版本 JVM 的数据宽度

```
if (_major_version == 45 && _minor_version <= 2) {
    cfs->guarantee_more(4, CHECK_(nullHandle));
    max_stack = cfs->get_u1_fast();
    max_locals = cfs->get_u1_fast();
    code_length = cfs->get_u2_fast();
} else {
    cfs->guarantee_more(8, CHECK_(nullHandle));
    max_stack = cfs->get_u2_fast();
    max_locals = cfs->get_u2_fast();
    code_length = cfs->get_u4_fast();
}
```

在 Java 方法的 code 属性中，紧跟在 `code_length` 之后的就是 Java 源码所对应的字节码指令了，这部分指令最终会被从 Java class 字节码文件复制到内存中，具体而言是复制到 Java 方法在 JVM 内部所对应的 `methodOop` 对象的内存区域。由于当前阶段仍在 Java 方法属性的解析阶段，尚未创建 `methodOop` 对象，因此在这一步不会进行复制。但是 HotSpot 却通过 `code_start = cfs->get_u1_buffer()` 将字节码的第一条指令在 Java class 字节码文件中的位置记录下来，保存到 `code_start` 变量中。在前面解析出的 `code_length` 最终将会在后续创建 `methodOop` 时作为参数传递进去，最终 HotSpot 将依据 `code_start` 和 `code_length` 这两个数据确定从 Java class 字节码文件中要复制的字节码指令区域。具体复制的方式及复制的目标位置在后文再讲解，此处先略过不提。

注意：HotSpot 调用 `cfs->get_u1_buffer()` 函数来获取第一条字节码指令在字

节码文件中的位置，而非该位置处的值。在 HotSpot 内部，获取字节码文件某个位置的值，通常调用诸如 `get_ul_fast()` 这样的方法。

8.2.2 LVT&LVTT

在 Java 方法的属性中，有一种属性是局部变量表——`LocalVariableTable`，在 JVM 内部简写为 LVT。

`LocalVariableTable` 属性用于描述 Java 方法栈帧中局部变量表中的变量与 Java 源代码定义的变量之间的关系，这种关系并非运行时必需，所以默认情况下不会生成到 class 文件中。若想生成到 class 字节码文件中，则可以通过在 `javac` 命令中使用 `-g:vars` 选项生成这项信息。

如果 Java class 字节码文件中没有生成局部变量表，则在调试 Java 程序时，无法看到源码中所定义的参数名称，IDE 可能使用 `arg0`、`arg1` 占位符替代原来的参数，这对程序运行没有任何影响，但会影响使用体验。

在 Java class 字节码文件中，`LocalVariableTable` 属性表的结构如表 8.1 所示。

表 8.1 `LocalVariableTable` 属性表结构

类 型	名 称	数 量
u2	<code>attribute_name_index</code>	1
u4	<code>attribute_length</code>	1
u2	<code>local_variable_table_length</code>	1
<code>local_variable_info</code>	<code>local_variable_table</code>	<code>local_variable_table_length</code>

注：表中的 `local_variable_info` 类型是一种特殊的复合数据结构，该复合结构用于描述 Java 方法栈帧与源代码中局部变量的关联，其结构如表 8.2 所示。

表 8.2 `local_variable_info` 类型的数据结构

类 型	名 称	数 量
u2	<code>start_pc</code>	1
u2	<code>length</code>	1
u2	<code>name_index</code>	1
u2	<code>descriptor_index</code>	1
u2	<code>index</code>	1

这 5 种属性的含义如下：

- ◎ `start_pc`，表示当前局部变量的生命周期开始的字节码偏移量。
- ◎ `length`，表示当前局部变量的作用范围覆盖长度，和 `start_pc` 一起就表示局部变量在字节码中的作用范围。
- ◎ `name_index`，当前局部变量的名称所对应的常量池的索引号。
- ◎ `descriptor_index`，当前局部变量的描述信息所对应的常量池的索引号。
- ◎ `index`，当前局部变量在栈帧局部变量中 `slot` 的位置，如果数据类型是 `long` 或 `double`（64 位），`slot` 的位置为 `index` 和 `index + 1`。

研究局部变量表对于 Java 应用程序开发者而言没有具体的意义，但是这关乎 Java 程序调试的一些工程实现策略，也就是说，这种策略不仅在 Java 中有应用，在其他编程语言中也有类似实现。在调试过程中，如何让 IDE 在面对编译后的毫无意义的栈帧占位符的背景下，能够为开发者呈现出这些占位符所对应的源码中的原始变量名，是所有编程语言都需要具备的能力。既然 HotSpot 开放源代码，我们不妨顺道研究一番，虽然这个话题也许与主题关联不是很大。

HotSpot 的局部变量表的分析逻辑在 Java 编译器中实现，Java 编译器会对 Java 源码进行语法解析，分析 Java 方法的栈帧结构，并据此分析各个变量的作用域。分析的结果被以特定的组织结构存储在 Java class 字节码文件中，具体就是表 8.1 和表 8.2 所列。在 HotSpot 加载某个 Java 类时，会按照这种特定的组织结构还原出编译期所分析的结果，从而在调试期间能够据此显示出局部变量的原始名称。在 HotSpot 中，局部变量表还原的逻辑封装在 `classFileParser.cpp::parse_localvariable_table()` 函数中，如下所示：

清单：/src/share/vm/classfile/classFileParser.cpp

作用：`parse_localvariable_table()` 解析局部变量表

```
u2* ClassFileParser::parse_localvariable_table(u4 code_length,
                                                u2 max_locals,
                                                u4 code_attribute_length,
                                                constantPoolHandle cp,
                                                u2* localvariable_table_length,
                                                bool isLVTT,
                                                TRAPS) {
    *localvariable_table_length = cfs->get_u2(CHECK_NULL);
    unsigned int size = (*localvariable_table_length) *
sizeof(Classfile_LVT_Element) / sizeof(u2);
    u2* localvariable_table_start = cfs->get_u2_buffer();
    if (!need_verify) {
        cfs->skip_u2_fast(size);
    } else {
        cfs->guarantee_more(size * 2, CHECK_NULL);
```



```

for(int i = 0; i < (*localvariable_table_length); i++) {
    u2 start_pc = cfs->get_u2_fast();
    u2 length = cfs->get_u2_fast();
    u2 name_index = cfs->get_u2_fast();
    u2 descriptor_index = cfs->get_u2_fast();
    u2 index = cfs->get_u2_fast();
    u4 end_pc = (u4)start_pc + (u4)length;
}
return localvariable_table_start;
}

```

这段逻辑很简单, 通过 `cfs->get_*_fast()` 函数从 Java class 字节码文件中读取连续的数据, 分别获取局部变量表的总长度、`start_pc`、`length`、`name_index` 等数据。不过这段逻辑仅仅是将这些数据读取出来, 进行了简单的校验便结束。真正的逻辑在 `classFileParser.cpp::parse_method()` 函数中:

清单: `/src/share/vm/classfile/classFileParser.cpp`

作用: `parse_method()` 复制局部变量表

```

//初始化 hash 表, 存储局部变量表, 校验局部变量表是否存在重复
initialize_hashtable(lvt_Hash);
Classfile_LVT_Element* cf_lvt;

//局部变量表信息存储在 constMethodOop 对象实例的内存区域的末尾, 在 byte code 之后
LocalVariableTableElement* lvt = m->localvariable_table_start();

for (tbl_no = 0; tbl_no < lvt_cnt; tbl_no++) {
    cf_lvt = (Classfile_LVT_Element *) localvariable_table_start[tbl_no];
    for (idx = 0; idx < localvariable_table_length[tbl_no]; idx++, lvt++) {

        //将局部变量表信息从 Java class 字节码文件中复制到 constMethodOop 对象实例的内存区域中
        copy_lvt_element(&cf_lvt[idx], lvt);

        //将当前局部变量表保存到 hash 表中, 若保存失败则抛异常, 表示有重复的局部变量表
        if (LVT_put_after_lookup(lvt, lvt_Hash) == false
            && _need_verify
            && _major_version >= JAVA_1_5_VERSION ) {
            clear_hashtable(lvt_Hash);
            classfile_parse_error("Duplicated LocalVariableTable attribute "
                                   "entry for '%s' in class file %s",
                                   cp->symbol_at(lvt->name_cp_index)->as_utf8(),
                                   CHECK_(nullHandle));
        }
    }
}
}

```

FileParser.cpp::parse_method()函数的这段逻辑，将局部变量表从 Java class 字节码文件复制到 Java 方法在 JVM 内部所对应的 constMethodOop 这个内部对象的内存区域，这个对象的内存布局结构在下文介绍。

在 JVM 运行期，在方法中打上断点，当 JVM 运行到断点处会执行中断而暂停，此时 JVM 能够通过栈帧上指向 methodOop 的指针定位到对应的 constMethodOop，由于局部变量表就保存在 constMethodOop 的内存区域的末尾位置，因此 JVM 能够基于 constMethodOop 进一步获取当前 Java 方法所有的局部变量表，从而在 IDE 中将方法入参和局部变量以原始的变量名显示出来。

在编译 Java 类时，是否启用生成局部变量表并放到字节码文件的选项，不仅会影响 IDE 调试时的体验，还会影响 Java 类库使用方的体验。在提供 class 文件给第三方使用时（不包含 Java 类源码），如果没有开启 -g:vars 选项，不生成局部变量表，则第三方将不会看到 Java 方法入参的原始名称。例如下面这个类：

清单：/Test.java

作用：演示字节码文件的局部变量表

```
public class Test{
    private long l;

    public void add(int aaa, int bbb){
        int z = aaa + bbb;
        long y = z + 5;
        int x = 3 + z;
    }

    public static void main(String[] args)throws Exception{
        Test test = new Test();
        test.add(2, 3);
    }
}
```

使用 javac 命令进行编译，不带 -g:vars 选项，使用 eclipse 打开编译后生成的 class 文件，内容如下：

清单：/Test.class

作用：演示字节码文件的局部变量表

```
public class Test {
    private long l;

    public Test() {
    }
}
```

```

public void add(int var1, int var2) {
    int var3 = var1 + var2;
    long var4 = var3 + 5;
    int var6 = 3 + var3;
}

public static void main(String[] var0) throws Exception {
    Test var1 = new Test();
    var1.add(2, 3);
}
}

```

可以看到,无论是 add()方法还是 main()方法,其内部的变量名都变成了以 var 开头的名字,这是 IDE 自己的命名。IDE 根据字节码指令进行逆向编译时,只能分析出 java 方法包含几个入参和几个局部变量,但是由于字节码文件中没有存储变量名,因此 IDE 无法还原出源码中真实的变量名。

注意观察 IDE 自动生成的这几个 var 变量名,2 个入参被命名为 var1 和 var2,3 个局部变量被分别命名为 var3、var4 和 var6。中间貌似有不连贯的地方,缺失了 var5。IDE 为何要跳过 var5 直接将最后一个变量命名为 var6 呢?不难猜测,IDE 对变量进行自动命名的规则是 var 拼接上入参或变量所对应的 slot 索引号。由于 add()是 Java 类成员方法,因此其第 1 个入参是隐藏的 this,所以其源码中显式声明的两个入参才分别被命名为 var1 和 var2。而 add()方法中的变量 y 的类型是 long, JVM 规范规定 long 类型的数据在 slot 中占用 2 个槽位,所以最后一个变量 x 的 slot 索引号自然就变成了 6。再观察 main()方法,由于这是一个静态方法,所以并不包含 this 这个隐藏的入参,所以 main()方法中显式声明的第一个入参 String[]被 IDE 自动命名成 var0。这里并不是要各位道友去研究 Java 逆向编译工程原理,只是要你明白,处处留心皆学问。

而带上 -g:vars 选项进行 javac 编译后,使用 eclipse 打开编译后的 class 文件,所看到的方法内部的变量名与 Java 源码中的变量名完全一致。之所以会这样,是因为使用 -g:vars 选项进行编译时,Java 方法内部的变量名称也会在字节码文件的常量池中存储,这是必须的,因为局部变量表中并没有直接存储变量名,而是将变量名存储到常量池并引用常量池的索引号。

下面举例对局部变量表加以说明,Java 示例便是上面所举的 Test 类,以 Test.add()方法作为分析对象。使用 -g:vars 选项对 Test 类进行编译,并使用 javap 命令分析字节码文件,得到 add()方法的分析结果如下:

```

public void add(int, int);
Code:
    Stack=2, Locals=7, Args_size=3
    0:  iload_1
    1:  iload_2
    2:  iadd

```

```

3:  istore_3
4:  iload_3
5:  iconst_5
6:  iadd
7:  i2l
8:  lstore 4
10: iconst_3
11: iload_3
12: iadd
13: istore 6
15: return

```

LocalVariableTable:

Start	Length	Slot	Name	Signature
0	16	0	this	LTest;
0	16	1	aaa	I
0	16	2	bbb	I
4	12	3	z	I
10	6	4	y	J
15	1	6	x	I

先看局部变量表中的 this、aaa 和 bbb 这 3 个人参，Start 都是 0，Length 都是 16。这里的 0 表示这 3 个参数的作用域从第 1 个字节码指令就开始生效，length 为 16 是因为 add() 方法字节码指令的总长度为 16，对于 Java 方法的入参，在方法内部任何地方都能引用到，所以其作用域的长度自然等于整个字节码指令的总长度。

接着看变量 z，其 Start 为 4，Length 为 12。在 add() 方法中，变量 z 的声明与初始化被合二为一了，其声明语句为 `int z = aaa + bbb`，其对应的字节码指令包含 4 条，分别是列表中 bci (byte code index，字节码指令偏移量) 等于 0、1、2、3 的 4 条指令。在这 4 条指令之后的所有字节码指令都能引用变量 z，所以变量 z 的作用域就从 4 开始，而其作用范围自然是后续所有字节码指令的长度，为 $(16-4)=12$ 。

同样的机制，变量 y 的声明语句的字节码指令到 bci=8 的位置结束，bci=8 的指令是 `lstore 4`，一共占 2 字节长度，所以下一条字节码指令的 bci 为 10，这正是变量 y 的作用域开始的位置，所以变量 y 的 Start=10。

JDK 1.5 引入了泛型之后，为 LocalVariableTable 属性添加了一个“姐妹属性”：LocalVariableTypeTable。在 JVM 内部，这个属性简称为 LVTT，该属性的结构和 LocalVariableTable 相似，仅仅把记录的字段描述符的 descriptor_index 替换成字段特征签名 (signature)。限于篇幅，本书不再描述该属性，有兴趣的道友可以自行研究。

8.3 创建 methodOop

完成对 Java 方法的各项属性解析之后, HotSpot 开始在内存中创建一个与 Java 方法对等的内部对象——methodOop。methodOop 包含 Java 方法的一切信息, 例如方法名、返回值类型、入参、字节码指令、栈深、局部变量表、行号表等。其实一言以蔽之, HotSpot 通过 methodOop, 将 Java class 字节码文件中的方法信息存储到了内存中, 并且这片内存区域是结构化的, 使得可以在 JVM 运行期方便地访问 Java 方法的各种属性信息。

ClassFileParser::parse_method()函数中通过调用 oopFactory::new_method()函数完成 methodOop 对象的创建, 且看 oopFactory::new_method() 的实现:

清单: /src/share/vm/memory/ooFactory.cpp

作用: new_method()逻辑

```
methodOop oopFactory::new_method(int byte_code_size, AccessFlags access_flags,
                                  int compressed_line_number_size,
                                  int localvariable_table_length,
                                  int checked_exceptions_length,
                                  bool is_conc_safe,
                                  TRAPS) {
    methodKlass* mk = methodKlass::cast(Universe::methodKlassObj());
    assert(!access_flags.is_native() || byte_code_size == 0,
           "native methods should not contain byte codes");
    constMethodOop cm = new_constMethod(byte_code_size,
                                          compressed_line_number_size,
                                          localvariable_table_length,
                                          checked_exceptions_length,
                                          is_conc_safe, CHECK_NULL);
    constMethodHandle rw(THREAD, cm);
    return mk->allocate(rw, access_flags, CHECK_NULL);
}
```

oopFactory::new_method()函数里面实际上创建了两个对象, 分别是 methodOop 和 constMethodOop。其中 methodOop 通过调用(methodKlass*)->allocate()函数创建, 而 constMethodOop 通过调用 oopFactory::new_constMethod()函数创建。

这里需要说明一下 constMethod 和 methodOop 的关系。无论在 JDK 6 还是最新的 JDK 8 中, 都保留了这两个对象。这两个对象的作用不同, methodOop 主要存储 Java 方法的名称、签名、访问标识、解释入口等信息, 而 constMethodOop 则用于存储方法的字节码指令、行号表、异常表等信息。

对于 JDK 6 和 JDK 8 而言, 这两者有微小的变化。在 JDK 6 中, Java 方法的最大栈深度和

局部变量表数量存储在 `methodOop` 中,而到了 JDK 8,这两个数据则被存储到了 `constMethodOop` 中。这种变化仅影响到 HotSpot 为 Java 方法创建栈帧时读取这两个变量的逻辑。

创建 `methodOop` 的逻辑如下:

清单: `/src/share/vm/oops/methodKlass.cpp`

作用: `allocate()` 逻辑

```
methodOop methodKlass::allocate(constMethodHandle xconst,
                                AccessFlags access_flags, TRAPS) {
    int size = methodOopDesc::object_size(access_flags.is_native());
    KlassHandle h_k(THREAD, as_klassOop());
    methodOop m = (methodOop)CollectedHeap::permanent_obj_allocate(h_k, size,
CHECK_NULL);
    assert(!m->is_parsable(), "not expecting parsability yet.");

    No_Safepoint_Verifier no_safepoint; // until m becomes parsable below
    m->set_constMethod(xconst());
    m->set_access_flags(access_flags);
    //...m 的一系列初始化

    xconst->set_method(m);
    return m;
}
```

前文曾经详细分析过常量池对象 `constantPoolOop` 的创建过程,在 `classFileParser::parseClassFile()` 方法中调用 `oopFactory::new_constantPool()` 函数创建常量池对象,后者调用 `constantPoolKlass::allocate()` 函数完成常量池的创建。

仔细比较 `constantPoolKlass::allocate()` 函数与这里的 `methodKlass::allocate()` 函数,会发现两者逻辑基本一致,都是先求取 `methodOop` 的大小 `size`,接着调用 `CollectedHeap::permanent_obj_allocate()` 函数在 perm 区(对于 JDK 8 而言则在 `metaSpace` 区)为所创建的对象分配内存。最后再调用一系列的 `setter()` 方法初始化所创建的对象。

创建 `methodOop` 的详细过程这里不再赘述,与 `constantPoolOop` 的创建基本一致。最终创建出来的 `methodOop` 对象的内存布局如图 8.2 所示(基于 JDK 6 的内存模型)。

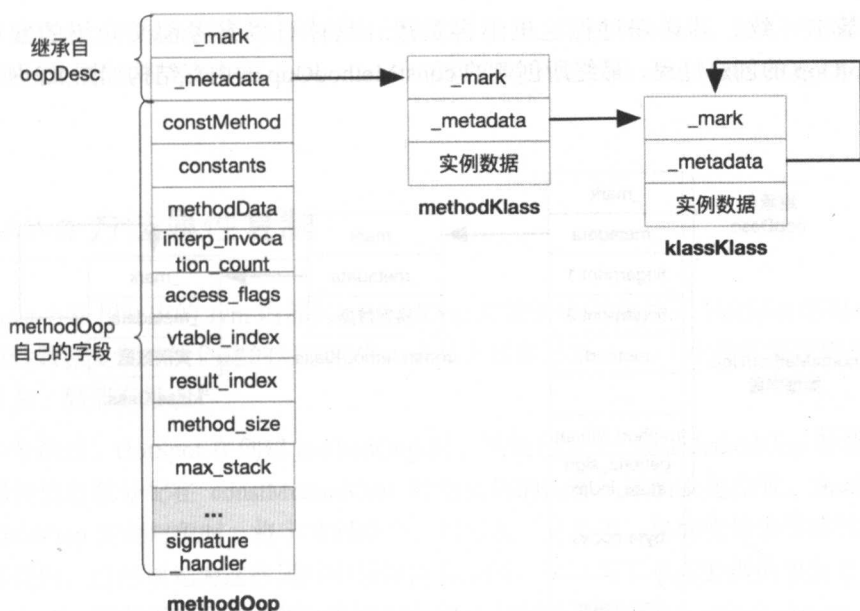


图 8.2 methodOop 对象的内存布局

虽然方法对象 `methodOop` 与常量池对象 `constMethodOop` 的创建机制基本相同，但是两者的数据结构仍然有所不同。`methodOop` 的内存结构与其类型定义的结构保持一致，而常量池 `constantPoolOop` 实例对象的内存的末尾还跟着常量池的元素数据。`HotSpot` 内部并没有为常量池的元素数组专门定义一种数据结构，这是因为不同 Java 类编译后所得到的常量池数组大小不同，并且各个元素成员的内存也完全不同，无法抽象成专门的数据结构，所以 `HotSpot` 只能将其分配到 `constantPoolOop` 实例对象的内存的末尾区域。但是 `methodOop` 与 `constantPoolOop` 一样，类型本身的字段并不足以保存 Java 方法的全部信息，例如 Java 方法的字节码指令、行号表等信息，在 `methodOop` 类型中并没有专门的字段存储这些信息，所以按理说 `methodOop` 也应该像 `constantPoolOop` 那样，将这些信息分配到 `methodOop` 实例对象的内存的末尾区域，但是 `methodOop` 并没有这么做。这是因为 `HotSpot` 为此专门另外定义了一种数据结构——`constMethodOop`，`HotSpot` 将 Java 方法的字节码指令及行号表等信息分配到了 `constMethodOop` 实例对象的内存的末尾区域。

在 `classFileParser::parseClassFile()` 函数调用 `oopFactory::new_method()` 函数创建 Java 方法对象的过程中，后者调用 `oopFactory::new_constMethod()` 函数完成 `constMethodOop` 的创建。在 `oopFactory::new_constMethod()` 函数中实际是调用 `constMethodKlass::allocate()` 函数完成 `constMethodOop` 的创建。`constMethodKlass::allocate()` 函数与 `constantPoolKlass::allocate()` 函数

的机制也基本一致，其详细过程这里不再赘述，具体可以参考前文介绍的常量池对象 `constantPoolOop` 的创建过程。最终所创建的 `constMethodOop` 的内存结构如图 8.3 所示（基于 JDK 6）。

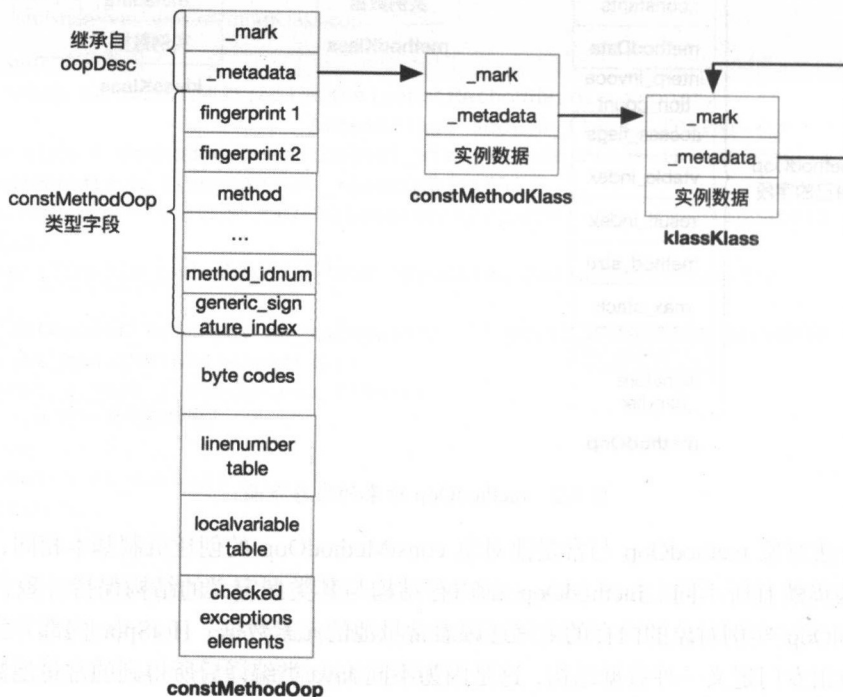


图 8.3 `constMethodOop` 内存布局

当 `oopFactory::new_constMethod()` 函数调用 `constMethodKlass::allocate()` 函数时，`constMethodKlass::allocate()` 函数的入参如下：

```

constMethodOop constMethodKlass::allocate(int byte_code_size,
                                           int compressed_line_number_size,
                                           int localvariable_table_length,
                                           int checked_exceptions_length,
                                           bool is_conc_safe,
                                           TRAPS)

```

可以看到，`constMethodKlass::allocate()` 函数入参包含字节码大小、行号表大小、局部变量表大小等信息，在 `constMethodKlass::allocate()` 函数内部所计算出的内存大小，是 `constMethodOop` 本身的大小与字节码大小、行号表大小等的总和，因为 HotSpot 将字节码指令、行号表、局部变量表等信息分配在 `constMethodOop` 对象实例的内存的末尾区域，所以在创建 `constMethodOop`

对象时,就将末尾区域所需要的内存提前申请和分配好。后续 HotSpot 会执行逻辑,将字节码指令、行号表、异常表等数据从 Java class 字节码文件中复制到 constMethodOop 末尾的这段内存中。

8.4 Java 方法属性复制

与 Java 方法相对应的 Jvm 内部的 methodOop 对象创建完成之后,HotSpot 需要将 Java 方法的几大属性数据复制进所创建的对象之中,这几大属性是指 code (主要是字节码指令)、行号表、异常表、局部变量表等。

上一节讲过,HotSpot 在创建 methodOop 时,顺便创建了 constMethodOop 对象实例,Java 方法的属性信息被分配在 constMethodOop 对象实例的内存区域的末尾位置。HotSpot 在创建 constMethodOop 实例对象时,将字节码指令、行号表、异常表、局部变量表等属性所需的空间大小计算在内,已经预先为这些属性申请好内存空间,所以接下来需要做的事就是将这些属性信息从 Java class 字节码文件中复制到申请的内存中。这段逻辑位于 ClassFileParser::parse_method() 函数中,如下:

清单: /src/share/vm/classfile/classFileParser.cpp

作用: 复制 Java 方法属性的逻辑

```
//...
//复制异常表
m->set_exception_table(exception_handlers());

//复制字节码指令
m->set_code(code_start);

//复制行号表
if (linenumber_table != NULL) {
    memcpy(m->compressed_linenumber_table(),
           linenumber_table->buffer(), linenumber_table_length);
}
//...
```

这些逻辑大同小异,都是将前面步骤已经解析好的属性信息复制到对应的内存位置,其中局部变量表的复制逻辑已经在前文讲解 LVT 时进行过描述。这里重点关注字节码指令的处理,相信这也是很多道友所关心的问题。

字节码指令的处理主要是调用 m->set_code(address code_start)函数,入参 code_start 在前置流程中已经解析出来,其值是当前 Java 方法的第一条字节码指令在读入内存中的字节码文件流

中的内存位置。

methodOop::set_code(address)函数逻辑如下：

清单：/src/share/vm/oops/methodOop.hpp

作用：复制 Java 方法字节码指令

```
void set_code(address code) {  
    return constMethod()->set_code(code);  
}
```

本函数里面调用了 constMethod()->set_code(address)函数，该函数逻辑如下：

清单：/src/share/vm/oops/constMethodOop.hpp

作用：复制 Java 方法字节码指令

```
void set_code(address code) {  
    if (code_size() > 0) {  
        memcpy(code_base(), code, code_size());  
    }  
}
```

由此可见，constMethodOop::set_code(address)函数最终调用了 memcpy()函数，memcpy()函数有 3 个形式入参，分别表示目的内存首地址、复制源内存首地址、复制长度。在复制 Java 方法的字节码指令时，第 2 个入参 code 已经包含明确的值，就是当前 Java 方法的第一条字节码指令在读入内存中的字节码文件流中的内存位置。而 constMethodOop::set_code(address)函数第一个入参是 code_base()函数的返回值，该函数逻辑如下：

清单：/src/share/vm/oops/constMethodOop.hpp

作用：复制 Java 方法字节码指令

```
address code_base() const {  
    return (address) (this+1);  
}
```

address 是指针类型，因此 this+1 表示 constMethodOop 实例对象的末尾位置的下一位，这正是 Java 方法字节码指令的存放位置。

到此为止，Java 方法字节码指令复制的实现逻辑便清楚了，字节码指令最终从 Java class 字节码文件中复制到了 constMethodOop 对象实例的内存的末尾位置。在 Java 程序运行期，HotSpot 将根据所调用的目标函数，找到该目标 Java 方法在内存中所对应的 methodOop 对象实例，并根据 methodOop 对象实例找到对应的 constMethodOop，最终基于 constMethodOop 定位到目标 Java 方法所对应的字节码指令，并将首个字节码指令的内存位置保存到目标 Java 方法的栈帧中，HotSpot 通过 JMP 硬件指令跳转到这个位置开始执行 Java 方法所对应的字节码指令，从而完成 Java 方法逻辑。

8.5 <clinit>与<init>

1. 初识<clinit>与<init>

在 Java 中, 有两种特殊的方法, 分别是<clinit>和<init>。这两个方法并非由 Java 开发者所定义, 而是由 Java 编译器自动生成。当 Java 类中存在用 static 修饰的静态类型字段, 或者存在使用 static{} 块包裹的逻辑时, 编译器会自动生成<clinit>方法。而当 Java 类定义了构造函数, 或者其非 static 类成员变量被赋予了初始值时, 编译器会自动生成<init>方法。看下面 Java 示例:

清单: Test.java

作用: 演示<init>与<clinit>

```
public class Test{

    private Integer i = 3;
    private static int a = 90;

    public void add(int a, int b){
        Test test = this;
        int z = a + b;
        int x = 3;
    }

    public static void main(String[] args){
        Test test = new Test();
        test.add(2, 3);
    }
}
```

将该类编译后, 使用 javap 命令查看编译后的 class 文件, 显示如下:

```
$ javap -verbose Test
Compiled from "Test.java"
public class Test extends java.lang.Object
  SourceFile: "Test.java"
  minor version: 0
  major version: 50
  Constant pool:
const #1 = Method      #8.#24; // java/lang/Object."<init>":()V
const #2 = Method      #25.#26; // java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
const #3 = Field       #4.#27; // Test.i:Ljava/lang/Integer;
//...
const #11 = Asciz      a;
const #12 = Asciz      I;
```

```

const #13 = Asciz  <init>;
const #14 = Asciz  ()V;
//...
const #19 = Asciz  main;
const #20 = Asciz  ([Ljava/lang/String;)V;
const #21 = Asciz  <clinit>;
const #22 = Asciz  SourceFile;
//...
const #33 = Asciz  valueOf;
const #34 = Asciz  (I)Ljava/lang/Integer;;

{
public Test();
    Code:
        Stack=2, Locals=1, Args_size=1
        0:  aload_0
        1:  invokespecial  #1; //Method java/lang/Object."<init>":()V
        4:  aload_0
        5:  iconst_3
        6:  invokestatic  #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
        9:  putfield      #3; //Field i:Ljava/lang/Integer;
       12:  return

public void add(int, int);
    Code:
        Stack=2, Locals=6, Args_size=3
        0:  aload_0
        //...

public static void main(java.lang.String[]);
    Code:
        Stack=3, Locals=2, Args_size=1
        0:  new #4; //class Test
        //...

static {};
    Code:
        Stack=1, Locals=0, Args_size=0
        0:  bipush 90
        2:  putstatic  #7; //Field a:I
        5:  return
}

```

通过 `javap` 命令可以看到，`Test` 类一共包含 4 个方法，其中 `main()` 和 `add()` 方法在 `Test` 类中被显式定义，然而另外 2 个方法却并未在 `Test` 类中进行过声明，分别是 `public Test()` 和 `static {}`，其实这 2 个方法便是由编译器自动生成的。由于 `Test` 类中有一个被 `static` 修饰的成员变量，因

此编译器自动生成了 `static {}` 这样的方法,同时由于 `Test` 类的成员变量 `i` 在声明时就被赋予了初值,因此编译器自动生成了 `public Test()` 这样的构造函数。

其实,这里的 `public Test()` 和 `static {}` 这两个方法,方法名分别是 `<init>` 和 `<clinit>`,使用 `javap` 命令解析 Java class 文件时,在常量池中便有这两个方法的名称,索引号分别为 13 和 21,如下:

```
const #13 = Asciz <init>;
const #21 = Asciz <clinit>;
```

只是 `javap` 命令并没有像 `main()` 和 `add()` 方法那样直接将 `<init>` 和 `<clinit>` 这 2 个方法的名称显示出来,而是以 `public Test()` 和 `static {}` 这样的形式加以显示。

2. 使用 HSDB 查看 `<clinit>` 和 `<init>`

通过 HSDB 这个神器,可以直观地显示出 `<init>` 和 `<clinit>` 这 2 个方法。使用 JDB 在 `Test.add()` 方法内部打上断点并启动调试,然后启动 HSDB 并使其连上 `Test` 进程,接着单击 HSDB 的 `Tool->Class browser` 工具按钮,选择 `Test` 类,就能查看 `Test` 类中的全部方法(关于 JDB 工具和 HSDB 工具的使用,前文已有详细介绍,这里不再赘述),如图 8.4 所示。

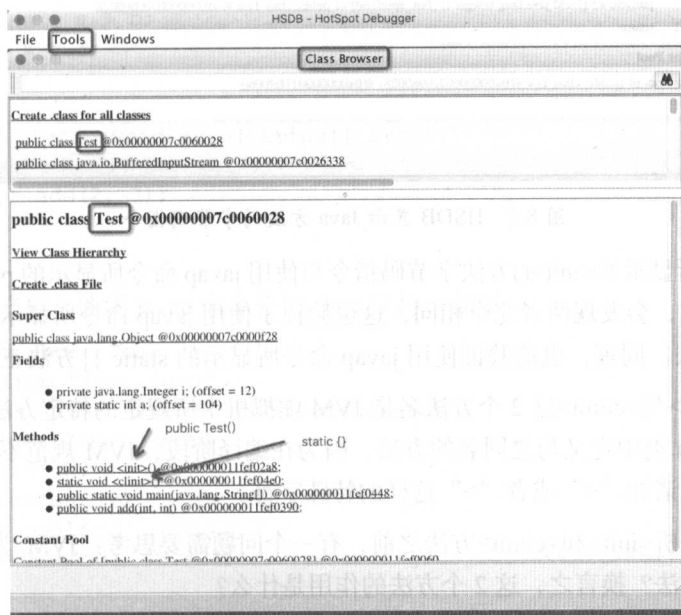


图 8.4 使用 HSDB 显示 `Test` 类的全部方法

图 8.4 显示, `Test` 类中的确存在名为 `<init>` 和 `<clinit>` 的两个方法,而这两个方法正对应于用 `javap` 命令所显示出来的 `public Test()` 和 `static {}` 方法。

在如图 8.4 所显示的 HSDB 的 Class Browser 视窗中单击某个方法，HSDB 会显示该方法的字节码指令，例如单击 public void <init>()方法名，显示的字节码指令如图 8.5 所示。

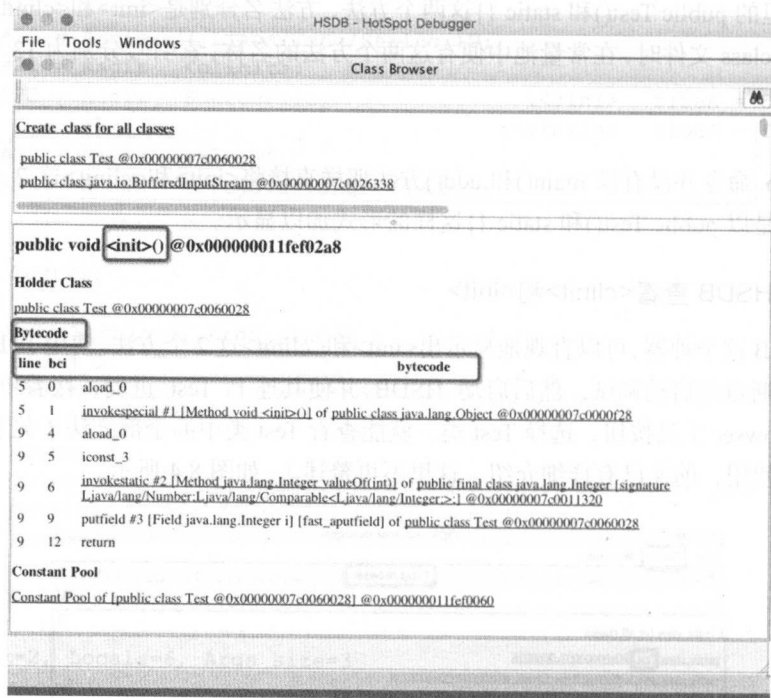


图 8.5 HSDB 显示 Java 方法的字节码指令

将 HSDB 中所显示的<init>()方法字节码指令与使用 javap 命令所显示的 public Test()方法字节码指令进行对比，会发现两者完全相同，这也验证了使用 javap 命令所显示的 public Test()方法正是<init>()方法。同理，也能验证使用 javap 命令所显示的 static {}方法正是<clinit>()方法。

事实上，<init>与<clinit>这 2 个方法名是 JVM 虚拟机中所规定的特定方法名，Java 应用开发者并不能在 Java 类中定义与之同名的方法，因为在编译阶段，JVM 规范不允许 Java 源程序中的方法名中存在诸如 “<” 或者 “>” 这样的特殊字符。

在继续深入分析<init>和<clinit>方法之前，有一个问题需要思考：JVM 为什么需要编译器自动生成这 2 个方法？换言之，这 2 个方法的作用是什么？

其实<init>方法好说，等同于类的构造函数，因此在遇到诸如 new 等指令时，自然会调用该方法。而当 Java 类中存在 static 成员变量，或者存在 static {}包裹的代码块时，则 JVM 加载 Java 类时，会触发<clinit>方法，完成 static 成员变量的初始化，或者执行被 static {}包裹的代码段的逻辑。

3. <init>详解

在上面的 Test 类的例子中，并没有在 Test 类中显式定义构造函数，但是编译器还是默认生成了一个构造函数，那么如果开发者显式定义一个无参的构造函数，编译器会如何处理呢？且举一例进行说明，将上述 Test 类实例稍微做下修改，添加一个无参的构造函数，修改后的 Test 类如下：

清单：Test.java

作用：演示<init>与<clinit>

```
public class Test{
    private Integer i = 3;
    private static int a = 90;

    public Test(){
        short s = 8;
    }

    public void add(int a, int b){
        Test test = this;
        int z = a + b;
        int x = 3;
    }

    public static void main(String[] args){
        Test test = new Test();
        test.add(2, 3);
    }
}
```

编译 Test 类之后，使用 javap 命令分析 class 文件，如下：

```
$ javap -verbose Test
Compiled from "Test.java"
public class Test extends java.lang.Object
  SourceFile: "Test.java"
  minor version: 0
  major version: 50
  Constant pool:
const #1 = Method      #8.#24; // java/lang/Object."<init>":()V
//...
const #33 = Asciz      valueOf;
const #34 = Asciz      (I)Ljava/lang/Integer;;

{
  public Test();
  Code:

```

```

Stack=2, Locals=2, Args_size=1
0:  aload_0
1:  invokespecial  #1; //Method java/lang/Object."<init>":()V
4:  aload_0
5:  iconst_3
6:  invokestatic  #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
9:  putfield      #3; //Field i:Ljava/lang/Integer;
12: bipush 8
14: istore_1
15: return

```

```
public void add(int, int);
```

```
Code:
```

```
//...
```

本次为 Test 类显式定义了一个无参的构造函数,这种改动主要影响<init>()方法。使用 javap 命令分析 Test.class 文件时,只需要观察 public Test()这部分的字节码指令的变化。在上面这段字节码指令中,能够看到如下指令:

```

5:  iconst_3
6:  invokestatic  #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
9:  putfield      #3; //Field i:Ljava/lang/Integer;
12: bipush 8
14: istore_1

```

其中 bci 等于 5、6、9 的字节码指令对应的 Java 源码是 Test 类的成员变量 i 的定义和赋值: private Integer i = 3。接着 bci 等于 12 和 14 的字节码指令对应的 Java 源码是 Test 构造函数中的 short s=8。由此可见,即使人为定义了默认的无参构造函数,编译器仍然会修改构造函数的字节码指令,将 Java 类成员变量的初始化指令都插入到构造函数<init>()方法的字节码指令中。这一点是<init>()方法与其他人为定义的 Java 类方法的最大之不同,对于人为定义的 Java 类方法,其字节码指令一定与 Java 源码保持一致,编译器不会随便往里加入其他逻辑。

前文讲过,当 Java 类的成员变量存在初始化行为时,其初始化的逻辑就会被嵌入到<init>()方法中。其实,还有一种情况也会如此,那就是在 Java 类中使用 {} 包裹的代码逻辑,也会被嵌入到<init>()方法中。修改 Test 类,如下:

清单: Test.java

作用: 演示<init>与<clinit>

```

public class Test{
    private Integer i = 3;

    {
        System.out.println("i=" + i);
    }
}

```

```

    public static void main(String[] args){
        Test test = new Test();
    }
}

```

编译后使用 javap 命令分析 Test.class 字节码文件，输出如下：

```

public Test();
  Code:
    Stack=3, Locals=1, Args_size=1
    0:   aload_0
    1:   invokespecial   #1; //Method MyClass."<init>":()V
    4:   aload_0
    5:   iconst_3
    6:   invokestatic     #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
    9:   putfield         #3; //Field i:Ljava/lang/Integer;
   12:  getstatic        #4; //Field java/lang/System.out:Ljava/io/PrintStream;
   15:  new               #5; //class java/lang/StringBuilder
   18:  dup
   19:  invokespecial     #6; //Method java/lang/StringBuilder."<init>":()V
   22:  ldc               #7; //String i=
   24:  invokevirtual     #8; //Method
java/lang/StringBuilder.append:(Ljava/lang/String;)Ljava/lang/StringBuilder;
   27:  aload_0
   28:  getfield         #3; //Field i:Ljava/lang/Integer;
   31:  invokevirtual     #9; //Method
java/lang/StringBuilder.append:(Ljava/lang/Object;)Ljava/lang/StringBuilder;
   34:  invokevirtual     #10; //Method
java/lang/StringBuilder.toString:()Ljava/lang/String;
   37:  invokevirtual     #11; //Method
java/io/PrintStream.println:(Ljava/lang/String;)V
   40:  return
  LineNumberTable:
    line 4: 0
    line 9: 4
    line 13: 12
    line 14: 40

```

可以看出，上面有一大段字节码指令在处理对应的 Test 类源码中被 {} 包裹的块逻辑中的 System.out.println("i=" + i)。

现在再对 Test 类进行微调，刚才显式定义了一个无参的默认构造函数，现在则定义含入参的非默认的构造函数，同时将刚才无参的默认构造函数删除，修改后的 Test 类如下：

清单：Test.java

作用：演示<init>与<clinit>

```
public class Test{
```

```
private Integer i = 3;
private static int a = 90;

public Test(int x){
    x = 12;
}

public void add(int a, int b){
    Test test = this;
    int z = a + b;
    int x = 3;
}

public static void main(String[] args){
    Test test = new Test(10);
    test.add(2, 3);
}
}
```

编译 Test 类并使用 javap 命令分析 Test.class 文件, 此时得到的字节码指令如下:

```
$ javap -verbose Test
Compiled from "Test.java"
public class Test extends java.lang.Object
  SourceFile: "Test.java"
  minor version: 0
  major version: 50
  Constant pool:
const #1 = Method   #8.#25; // java/lang/Object."<init>":()V
//...
const #35 = Asciz   valueOf;
const #36 = Asciz   (I)Ljava/lang/Integer;;

{
public Test(int);
  Code:
    Stack=2, Locals=2, Args_size=2
    0:   aload_0
    1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
    4:   aload_0
    5:   iconst_3
    6:   invokestatic     #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/
Integer;
    9:   putfield         #3; //Field i:Ljava/lang/Integer;
   12:   bipush 12
   14:   istore_1
   15:   return
}
```

```
public void add(int, int);
Code:
//...
```

可以看到,现在得到了 `public Test(int)` 方法及其字节码指令,其字节码指令中的 `bci` 等于 5、6、9 的这 3 条字节码指令仍然对应 `Test` 类中的 `private Integer i = 3` 这一条源代码,然后 `bci` 等于 12、14 的这两条字节码指令对应 `Test(int)` 这个含入参的构造函数中的 `x = 12` 这一句源代码。由此可见,当为 Java 类定义非默认的构造函数时,Java 编译器仍然会“擅自”修改构造函数的逻辑,将 Java 类成员的初始化逻辑所对应的字节码指令插入到构造函数所对应的字节码指令中。

到这里可以看出来, `<init>()` 方法还是有点意思的,看不见的背后,其实编译器为我们做了很多工作。不过 `<init>()` 方法还有更多值得玩味的特性。刚才分别为 `Test` 类显式定义了一个无参的默认构造函数和一个含入参的构造函数,现在看看如果为一个 Java 类同时定义两个乃至多个构造函数,编译器如何处理。修改 `Test` 类,变成如下:

清单: `Test.java`

作用: 演示 `<init>` 与 `<clinit>`

```
public class Test{
    private Integer i = 3;
    private static int a = 90;

    public Test(){
        short s = 8;
    }

    public Test(int x){
        x = 12;
    }

    public void add(int a, int b){
        Test test = this;
        int z = a + b;
        int x = 3;
    }

    public static void main(String[] args){
        Test test = new Test();
        test.add(2, 3);
    }
}
```

可以看到,现在 `Test` 类包含 2 个显式定义的构造函数,编译后使用 `javap` 命令分析 `Test.class`

文件，显示如下：

```
$ javap -verbose Test
Compiled from "Test.java"
public class Test extends java.lang.Object
  SourceFile: "Test.java"
  minor version: 0
  major version: 50
  Constant pool:
const #1 = Method      #8.#25; //  java/lang/Object."<init>":()V
//...
const #35 = Asciz      valueOf;
const #36 = Asciz      (I)Ljava/lang/Integer;;

{
public Test();
  Code:
    Stack=2, Locals=2, Args_size=1
    0:   aload_0
    1:   invokespecial  #1; //Method java/lang/Object."<init>":()V
    4:   aload_0
    5:   iconst_3
    6:   invokestatic   #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
    9:   putfield       #3; //Field i:Ljava/lang/Integer;
   12:   bipush 8
   14:   istore_1
   15:   return

public Test(int);
  Code:
    Stack=2, Locals=2, Args_size=2
    0:   aload_0
    1:   invokespecial  #1; //Method java/lang/Object."<init>":()V
    4:   aload_0
    5:   iconst_3
    6:   invokestatic   #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
    9:   putfield       #3; //Field i:Ljava/lang/Integer;
   12:   bipush 12
   14:   istore_1
   15:   return

public void add(int, int);
  Code:
  //...
```

仔细观察使用 `javap` 命令得到的输出中的 `Test()` 和 `Test(int)` 这两个方法的字节码，会发现这两个方法都包含 `bci` 等于 5、6、9 的这 3 条字节码指令，并且这 3 条字节码指令的含义相同，

都对应 Test 类中的 `private Integer i = 3` 这一句源代码。而这两个方法中，在 bci 为 9 之后的字节码指令则不再相同，因为两个构造函数内部的逻辑不同。由此可见，当为 java 类定义多个构造函数时，编译器会将类成员变量的初始化逻辑的字节码指令编排到每一个构造函数的字节码指令中。而道理其实很简单，因为不管是通过 `new Test()` 去实例化 Test 类，还是通过 `new Test(10)` 去实例化 Test 类，所实例化出的 Test 类的成员变量 `i` 都应该具有初始值 3，所以编译器必须将类成员变量的初始化逻辑原封不动地复制到每一个构造函数里去。

现在转换下视角，从继承的角度看 `<init>()` 方法。让 Test 类继承一个父类 MyClass，MyClass 定义如下：

清单：MyClass.java

作用：演示 `<init>` 与 `<clinit>`

```
public abstract class MyClass {
    protected long plong = 12L;
}
```

修改 Test 类，使其继承 MyClass。为了节省篇幅，这里不贴出 Test 类。由于 Test 类继承了 MyClass 类，因此 Test 类便理所当然地继承了 MyClass 类中的成员变量 `plong`。当实例化 Test 类时，JVM 除了要完成 Test 类的成员变量的赋初值逻辑，也要完成继承自 MyClass 父类的成员变量的赋初值逻辑。因此，Java 编译器必然要将 MyClass 父类成员变量的初始化逻辑也嵌入到 Test 类的全部构造函数之中。

基于这种理论猜测来做验证——编译 Test 类，并使用 `javap` 命令分析 Test.class 字节码文件，`javap` 命令的输出如下：

```
$ javap -verbose Test
Compiled from "Test.java"
public class Test extends MyClass
  SourceFile: "Test.java"
  minor version: 0
  major version: 50
  Constant pool:
const #1 = Method      #8.#25; // MyClass."<init>":()V
//...
const #35 = Asciz      valueOf;
const #36 = Asciz      (I)Ljava/lang/Integer;;

{
  public Test();
  Code:
    Stack=2, Locals=2, Args_size=1
    0:   aload_0
    1:   invokespecial    #1; //Method MyClass."<init>":()V
```



```
4:  aload_0
5:  iconst_3
6:  invokestatic  #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
9:  putfield  #3; //Field i:Ljava/lang/Integer;
12:  bipush 8
14:  istore_1
15:  return
```

```
public Test(int);
```

```
Code:
```

```
Stack=2, Locals=2, Args_size=2
```

```
0:  aload_0
```

```
1:  invokespecial  #1; //Method MyClass."<init>":()V
```

```
4:  aload_0
```

```
5:  iconst_3
```

```
6:  invokestatic  #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
```

```
9:  putfield  #3; //Field i:Ljava/lang/Integer;
```

```
12:  bipush 12
```

```
14:  istore_1
```

```
15:  return
```

```
public void add(int, int);
```

```
Code:
```

```
//...
```

观察 `javap` 命令的输出, 在 `public Test()` 和 `public Test(int)` 这 2 个方法中, `bci=1` 的指令都是:

```
invokespecial  #1; //Method MyClass."<init>":()V
```

可见, 在 `Test` 类的两个构造函数中自动调用了父类的默认构造函数。Java 编译器将 `MyClass` 父类的成员变量的初始化逻辑封装到了 `MyClass` 父类自己的默认构造函数中, 并通过将调用父类默认构造函数的逻辑嵌入到子类的各个构造函数中, 从而在子类构造函数中完成父类成员变量的初始化逻辑。关于 `MyClass.<init>()` 方法, 各位道友可以自行使用 `javap` 命令分析 `MyClass.<init>()` 方法的字节码指令, 验证该方法有无嵌入 `MyClass` 类成员变量的初始化逻辑。

到了这里, 笔者有个担心, 如果父类有多个构造函数, 那么编译器会让子类的构造函数调用父类的哪个构造函数呢? 各位道友可以大胆猜测一把, 这里直接通过试验进行验证。修改上述 `MyClass` 类, 在其中定义两个构造函数, 修改后的 `MyClass` 类如下:

清单: `MyClass.java`

作用: 演示 `<init>` 与 `<clinit>`

```
public abstract class MyClass {
    protected long plong = 12L;

    public MyClass(){
```

```

        plong = (long)'C';
    }

    public MyClass(int x){
        x = 32;
    }
}

```

注意，上述类文件中为 MyClass 类定义了一个默认的无参构造函数，另一个则是含入参的非默认构造函数。

Test 类还是继承 MyClass 类，为节省篇幅，这里不贴出 Test 类。编译 Test 类并使用 javap 命令分析 Test.class 字节码文件，输出如下（且看子类 Test 的两个构造函数到底会调用父类的哪个构造函数）：

```

$ javap -verbose Test
Compiled from "Test.java"
//...

{
public Test();
    Code:
        Stack=2, Locals=2, Args_size=1
        0:   aload_0
        1:   invokespecial   #1; //Method MyClass."<init>":()V
        4:   aload_0
        5:   iconst_3
        //...

public Test(int);
    Code:
        Stack=2, Locals=2, Args_size=2
        0:   aload_0
        1:   invokespecial   #1; //Method MyClass."<init>":()V
        4:   aload_0
        5:   iconst_3
        //...

//...
}

```

javap 命令的输出显示，Test 的两个构造函数中被嵌入了调用 MyClass.<init>() 这个默认的构造函数的逻辑。

上面举了多个例子，全方位分析了 Java 类的 <init>() 方法（即构造函数）的生成规则，这些规则可以总结为如下几点：

- ◎ 无论一个 Java 类有无定义构造函数, 编译器都会自动生成一个默认的构造函数 `<init>()`。可以使用 `javap` 命令或者 `HSDB` 来验证。
- ◎ `<init>()` 方法主要完成 Java 类的成员变量的初始化逻辑, 同时会执行 Java 类中被 `{}` 所包裹的块逻辑。如果 Java 类中的成员变量没有被赋初值, 则在 `<init>()` 方法中不会对其进行初始化。
- ◎ 如果为 Java 类显式定义了多个构造函数, 无论是否是默认的非参构造函数, Java 编译器都会将 Java 类成员变量的初始化逻辑嵌入到每一个构造函数中, 并且嵌入的位置在各构造函数自身逻辑之前。
- ◎ 当 Java 类显式继承了父类时, 则 Java 编译器会让子类的各个构造函数调用父类的默认构造函数 `<init>()`, 从而在子类实例化时完成父类成员变量的初始化逻辑。
- ◎ 当父类中定义了多个构造函数时, 子类构造函数会调用父类默认构造函数。
- ◎ 子类构造函数对父类默认构造函数的调用顺序, 位于子类各个构造函数自身逻辑之前。

可以将上述 6 点总结成一句话:

一个类的各个构造函数的处理逻辑是, 调用父类默认构造函数, 完成类自身成员变量的初始化逻辑和被 `{}` 包裹的块逻辑, 调用各构造函数自身逻辑。

4. `<clinit>` 详解

前面详细分析了 `<init>()` 方法, 而 Java 编译器除了会自动生成 `<init>()` 方法, 还会自动生成 `<clinit>()` 方法。前文讲过, 当 Java 类中存在 `static` 字段, 或者被 `static {}` 包裹的代码逻辑时, 就会自动生成 `<clinit>()` 方法。

关于 `<clinit>()` 方法也有很多有趣的特性, 限于篇幅, 本书不一一分析, 各位道友可以按照前文分析 `<init>()` 方法的思路和角度来逐一分析 `<clinit>()` 方法的特性。这里仅举一复杂示例, 直接得出结论。

示例如下:

清单: `MyClass.java`

作用: 演示 `<init>` 与 `<clinit>`

```
public abstract class MyClass {  
    protected long plong = 12L;  
  
    static {  
        Integer i = 30;  
    }  
}
```

```

public class Test extends MyClass{
    private Integer i = 3;
    private static int a = 156;

    static{
        int y = 128;
    }

    public void add(int a, int b){
        Test test = this;
        int z = a + b;
        int x = 3;
    }

    public static void main(String[] args){
        Test test = new Test();
        test.add(2, 3);
    }
}

```

上述代码中定义了两个类，Test 类继承了 MyClass 类。编译 Test 类，先使用 javap 命令分析 MyClass.class 字节码文件，输出如下：

```

$ javap -verbose MyClass
Compiled from "MyClass.java"
public abstract class MyClass extends java.lang.Object
  SourceFile: "MyClass.java"
  minor version: 0
  major version: 50
  Constant pool:
const #1 = Method #7.#17; // java/lang/Object."<init>":()V
//...
const #25 = Asciz (I)Ljava/lang/Integer;;
{
//...

static {};
  Code:
    Stack=1, Locals=1, Args_size=0
    0: bipush 30
    2: invokestatic #5; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
    5: astore_0
    6: return
  LineNumberTable:
    line 11: 0

```

```
line 12: 6
}
```

可以看到, MyClass 的 static {} 方法中使用了 invokestatic 字节码指令完成变量 i 的初始化。

接着使用 javap 命令分析 Test.class 字节码文件, 输出如下:

```
$ javap -verbose Test
Compiled from "Test.java"
public class Test extends MyClass
  SourceFile: "Test.java"
  minor version: 0
  major version: 50
  Constant pool:
const #1 = Method      #8.#24; // MyClass."<init>":()V
//...
const #34 = Asciz      (I)Ljava/lang/Integer;;

{
//...

static {};
  Code:
    Stack=1, Locals=1, Args_size=0
    0:  sipush 156
    3:  putstatic #7; //Field a:I
    6:  sipush 128
    9:  istore_0
   10:  return
}
```

可以看到, Test 类的 static {} 方法中分别完成了成员变量 a 和局部变量 y 的初始化。由此可见, <clinit>() 方法不具有继承性, 原因其实也很简单, 因为 <clinit>() 方法是在类加载过程中被调用, 而父类与子类是分别加载的, 当父类加载完之后, 父类中的 static 成员变量初始化和被 static {} 所包裹的块逻辑已经执行完成, 没必要在子类加载时再执行一次, 所以子类只需完成自身 static 成员变量初始化以及被 static {} 所包裹的块逻辑即可。

5. init 与 clinit 执行顺序

<clinit>() 方法在 Java 类第一次被 Jvm 加载时调用, 而 <init>() 方法则在 Java 类被实例化时调用。由于类的加载一定位于类实例化之前, 因此 <clinit>() 方法一定在 <init>() 方法之前被触发。并且每一次实例化 Java 类都会调用 <init>() 方法, 而 <clinit>() 仅在类第一次加载时被调用, 以后再加载时不会重复调用。

下面举例说明。

清单: MyClass.java

作用: 演示<init>与<clinit>

```
public abstract class MyClass {
    protected long plong = 12L;

    {
        plong++;
        System.out.println("father.{}...plong=" + plong);
    }

    static {
        Integer i = 30;
        System.out.println("father.static{}...i=" + i);
    }

    public MyClass(){
        plong = (long)'C';
        System.out.println("father.constructor()...plong=" + plong);
    }
}

public class Test extends MyClass{
    private Integer i = 3;
    private static int a = 156;

    {
        i--;
        System.out.println("son.{}...i=" + i);
        System.out.println("son.{}...a=" + a);
    }

    static{
        a--;
        System.out.println("son.static{}...a=" + a);
    }

    public static void main(String[] args){
        Test test = new Test();
    }
}
```

上面定义了 MyClass 和 Test 两个类, Test 类继承自 MyClass 类。并且 Test 和 MyClass 类中都包含 {} 块逻辑和 static {} 块逻辑。运行 Test 类的 main() 方法, 输出如下:

```
father.static{}...i=30
```

```

son.static {...a=155
father. {...plong=13
father.constructor()...plong=67
son. {...i=2
son. {...a=155

```

根据该输出顺序可知，JVM 先加载了父类 MyClass，调用了父类的 static {} 块逻辑，因此首先输出“father.static {...i=30”。接着又加载了子类 Test，因此调用了子类的 {} 块逻辑，因此接着输出“son.static {...a=155”。接着实例化 Test 类，前文总结过，类的构造函数的执行顺序是“调用父类的默认构造函数->执行类成员变量初始化逻辑和被 {} 包裹的块逻辑->执行构造函数自身逻辑”，由于 Test 类继承了 MyClass 类，因此 Test 的构造函数先执行父类的构造函数。而父类 MyClass 中包含被 {} 包裹的块逻辑，因此父类构造函数先执行块逻辑，再执行自身逻辑，所以接下来的输出便是按照这种顺序执行的结果。

后续章节会从源码的角度分析 jvm 先执行类的加载逻辑再调用类的实例化的逻辑，此处先略过不提。

6. {} 和 static {} 的作用域

在 Java 类源码中可以使用 {} 和 static {} 来包裹块逻辑，前面也有多个例子进行了演示。不过这两者的作用域却是各不相同。

被 {} 所包裹的块，能够访问 Java 类的非静态成员变量和静态成员变量，但是其内部所定义的变量不能被外部所访问，这一点与 Java 类的普通函数（非 static）的作用域完全相同。而在编译阶段，就连 {} 块中的局部变量也与 Java 类中的非 static 方法的局部变量一样，被组织成局部变量表。且看下面示例：

清单：Test.java

作用：演示 {} 作用域

```

public class Test{
    private Integer i = 51;

    {
        int a = 1;
        int b = 2;
        int c = a + b;
    }
}

```

编译 Test 类，并使用 javap 命令分析 Test.class 字节码文件，输出的<init>()构造函数的字节码指令如下：


```

public Test();
Code:
    Stack=2, Locals=4, Args_size=1
    0:  aload_0
    1:  invokespecial  #1; //Method java/lang/Object."<init>":()V
    4:  aload_0
    5:  bipush  51
    7:  invokestatic  #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
    10: putfield    #3; //Field i:Ljava/lang/Integer;
    13:  iconst_1
    14:  istore_1
    15:  iconst_2
    16:  istore_2
    17:  iload_1
    18:  iload_2
    19:  iadd
    20:  istore_3
    21:  return

```

其中, bci 等于 4、5、7、10 的这 4 条字节码指令的作用是完成 Test 类成员变量 i 的初始化。而从 bci=13 的字节码指令开始, 执行 Test 类中被 {} 包裹的逻辑。注意 bci=14 的字节码指令是 istore_1, 这表示将数字 1 赋值给 {} 块中的 a 变量, 这正暗示出 Java 编译器将 {} 块中的变量 a 处理成了局部变量表中的第 2 个局部变量 (即第 2 个 slot)。同理, bci=16 和 bci=20 的两条字节码指令 istore_2 和 istore_3 也表明编译器将 {} 块中的第 2 和第 3 个局部变量 b 和 c 分别处理成了局部变量表中第 3 和第 4 个变量。

既然 Java 编译器将 {} 块中的局部变量处理成了局部变量表, 而局部变量表是函数相关的, 但是 {} 块逻辑并不是一个函数, 那么问题来了, 这里的局部变量表是谁的呢? 很显然, 由于 Java 编译器将 {} 块中的逻辑嵌入到了 <init>() 构造函数中, 因此这里的局部变量表自然就是 Java 类的构造函数的了。

不过当一个 Java 类中有多个 {} 块逻辑时, 问题将变得更加有趣。将上述 Test 类稍加改造, 如下:

清单: Test.java

作用: 演示 {} 作用域

```

public class Test{
    private Integer i = 51;

    {
        int a = 1;
        int b = 2;
        int c = a + b;
    }
}

```

```

{
    int x = 11;
    int y = 12;
    int z = x + y;
}

```

现在 Test 类中有两个 {} 块逻辑。前面刚刚讲过，Java 编译器会将 {} 块逻辑中的局部变量处理成构造函数中的局部变量表，现在有两个块逻辑，那么一个关键的问题就随之出现了：这两个 {} 块逻辑中的变量在局部变量表中的编号会是怎样的？是否相关？

带着这个疑问，编译 Test 类并使用 javap 命令分析 Test.class 字节码文件，输出的 <init>() 构造函数的字节码指令如下所示：

```

public Test();
Code:
    Stack=2, Locals=4, Args_size=1
    0:  aload_0
    1:  invokespecial  #1; //Method java/lang/Object."<init>":()V
    4:  aload_0
    5:  bipush  51
    7:  invokestatic  #2; //Method java/lang/Integer.valueOf:(I)Ljava/lang/Integer;
   10:  putfield     #3; //Field i:Ljava/lang/Integer;
   13:  iconst_1
   14:  istore_1      //int a=1
   15:  iconst_2
   16:  istore_2      //int b=2
   17:  iload_1
   18:  iload_2
   19:  iadd
   20:  istore_3      //int c=a+b
   21:  bipush  11
   23:  istore_1      //int x=11
   24:  bipush  12
   26:  istore_2      //int y=12
   27:  iload_1
   28:  iload_2
   29:  iadd
   30:  istore_3      //int z=x+y
   31:  return

```

上面将字节码指令与 Java 代码之间做了一一映射，Test 类中的 2 个 {} 块逻辑代码所对应的字节码指令都被嵌入到构造函数之中。注意观察这 2 个 {} 块逻辑中的局部变量在局部变量表中的编号（slot 索引），都是从 1 开始，也即第一个 {} 块逻辑中的 a 变量的 slot 索引是 1，而第二

个{}块逻辑中的 b 变量的 slot 索引也是 1。相应地,这 2 个块逻辑中的后续变量的 slot 索引编号都从 1 开始递增。由此可见,当 Java 类中存在多个{}块逻辑时,Java 编译器并不是简单地将其合并到构造函数之中,各个块中的局部变量的 slot 索引之间并不存在任何关联,唯一存在关联的就是各个{}块逻辑中的第一个局部变量的 slot 索引号相同。其实道理也很简单,由于 Java 类中的{}块就相当于一个独立的普通的 Java 函数,因此多个{}块中的局部变量之间彼此没有任何联系,相互不能引用,所以即使 Java 编译器将多个{}块逻辑都统一嵌入到构造函数之中,但是仍然保留了各个{}块逻辑的独立性。{}块中的局部变量既不能被其他{}块引用,更不会被构造函数引用到,因此当{}块逻辑执行完了之后,其局部变量表立即就被回收,被回收的局部变量表空间便被其他{}块的局部变量表所复用(其实并没有回收的动作,被复用就是被回收)。

讲完了{}块,接下来就轮到 static {}上场了。既然{}块可以被当作一个普通的 Java 方法处理,那么 static {}也自然可以被当作一个 static 修饰的 Java 静态方法处理,而事实上 JVM 也是这么规定的,所唯一不同的只是 static {}块逻辑是在 Java 类被加载时就执行,这是与其他 Java 类中静态方法的最大不同。

既然 static {}块可以被当作一个 Java 静态方法处理,那么这个块的作用域自然也就十分明了了:

- ◎ 仅能访问 Java 类中的静态成员变量,而不能访问非静态成员变量。
- ◎ static {}块中所定义的局部变量不能被外部访问。

关于 static {}的以上这两条特性,诸位道友可以自行写示例程序进行验证,这里就不浪费纸张了。

7. <init>与<clinit>的访问标识

<init>()方法包括其一系列的重载方法(即 Java 类中所定义的非默认构造函数),直接被编译器处理成 public 类型,这从前文一系列 Java 示例程序的 javap 命令的输出中可以看出。同时由于是构造函数,本身就没有返回类型。

而对于<clinit>()方法,在 HotSpot 源码中的 classFileParser.cpp::parse_method()方法中存在如下逻辑:

清单: /src/share/vm/classfile/classFileParser.cpp

作用: parse_method()设置<clinit>()方法签名

```
AccessFlags access_flags;
if (name == vmSymbols::class_initializer_name()) {
    if (_major_version < 51) { // backward compatibility
        flags = JVM_ACC_STATIC;
    } else if ((flags & JVM_ACC_STATIC) == JVM_ACC_STATIC) {
```

```

        flags &= JVM_ACC_STATIC | JVM_ACC_STRICT;
    }
} else {
    verify_legal_method_modifiers(flags, is_interface, name, CHECK_(nullHandle));
}

```

这段代码中的 `vmSymbols::class_initializer_name()` 就返回 `<clinit>`。通过这段源码可知，当 HotSpot 所解析的当前 Java 方法名是 `<clinit>` 时，就直接设置其访问标识是 `ACC_STATIC`。

8.6 查看运行时字节码指令

Java 方法经编译后就生成了字节码指令，在运行期字节码指令会被加载到 JVM 内存中，使用 HSDB 可以观察运行期的字节码指令，相信各位道友对此也非常感兴趣。

先准备一个示例程序：

清单：/Test.java

作用：观察运行期字节码指令

```

public class Test{
    private Integer i = 51;

    public void add(int a, int b){
        Test test = this;
        int z = a + b;
        int x = 3;
    }

    public static void main(String[] args){
        Test test = new Test();
        test.add(2, 3);
    }
}

```

使用 JDB 在 `Test.add()` 方法处打上断点并调试运行 Test 程序，运行至断点处，然后中断程序，接着使用 HSDB 连接上 Test 进程，观察内存。

注意，使用 JDB 调试 Test 程序时，必须加上 `-XX:-UseCompressedOops` 选项，这样迫使 JVM 放弃默认的指针压缩功能，否则无法显示内存对象的真实地址，给内存观察带来不便。

字节码指令被分配在 `constMethodOop` 对象的内存区域的末尾，因此只要在 HSDB 中能够查到 `add()` 方法所对应的 `constMethodOop` 所在的内存位置，就能基于此定位到字节码指令的内存位置，并进一步查看内存中的字节码指令长成啥样。

在 HSDB 中单击工具栏的 Tools->Class Browser 按钮，搜索 Test 类，可以看到该类的所有方法，如图 8.6 所示。

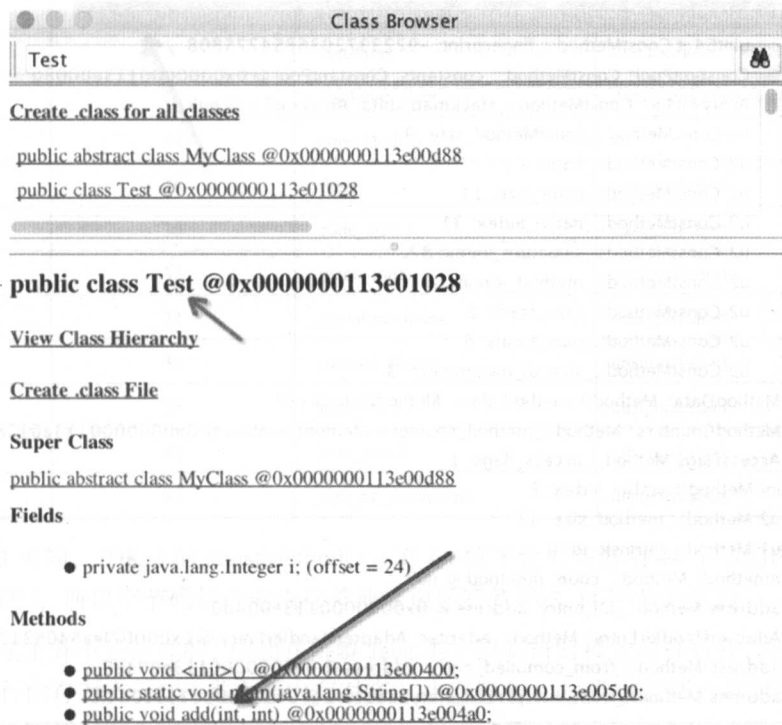


图 8.6 在 HSDB 中查看 Test 类的所有方法

图 8.6 中显示出 Test.add() 方法，HSDB 显示出该方法的签名和地址，其地址是 0x0000000113e004a0。

接着单击 HSDB 工具栏的 Tools->Inspector 按钮，输入 Test.add() 方法的地址并回车。Inspector 工具将分析 Test.add() 方法在 JVM 内部所对应的 MethodOop 对象实例的内存结构以及各个字段的内存地址，其中就包括我们关心的 constMethodOop 字段的内存地址。如图 8.7 所示。

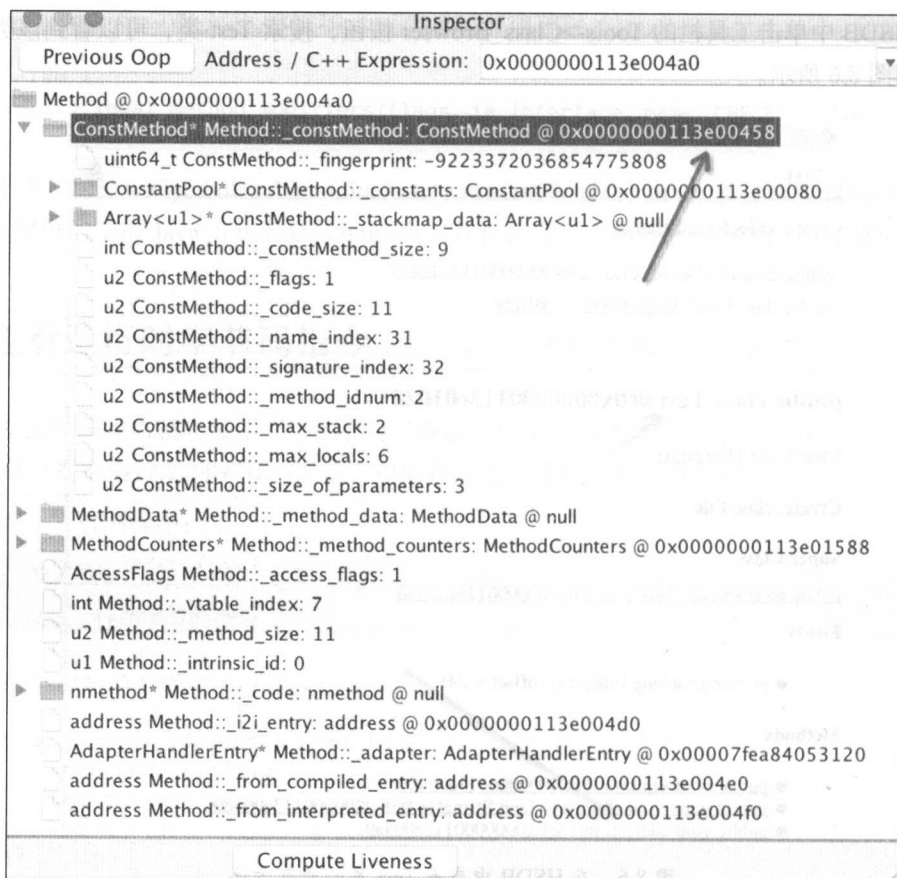


图 8.7 使用 HSDB 分析 Java 方法的内部结构

由于 MethodOop 类型包含 constMethodOop 的引用，因此在图 8.7 中可以看到 constMethodOop 字段的内存地址，该地址为 0x0000000113e00458。这个地址正是 Test.add() 方法在 JVM 内部所对应的 constMethodOop 对象实例的内存地址，而 Test.add() 方法的字节码指令就跟在 constMethodOop 对象实例的内存后面。HSDB 没有直接给出字节码指令的查看工具，因此只能通过计算字节码指令的起始内存地址并使用 Mem 工具去查看字节码指令。字节码指令的起始内存地址的计算很简单，只需要使用 constMethodOop 的内存首地址加上该对象实例所占内存空间的大小，就得到字节码指令的内存首地址。

本示例运行在 64 位平台上，并且使用 JDB 工具进行调试时添加了 -XX:-UseCompressedOops 选项，因此 constMethodOop 的所有指针类型的字段都占用 8 字节的内存空间，constMethodOop 对象内各个字段的偏移量如表 8.3 所示（基于 JDK 8）。

表 8.3 constMethodOop 字段偏移量

占用内存	偏 移	字 段	类 型
8	0	_fingerprint	volatile unsigned __int64
8	8	_constants	ConstantPool *
8	16	_stackmap_data	Array<unsigned char> *
4	24	_constMethod_size	int
2	28	_flags	unsigned short
2	30	_code_size	unsigned short
2	32	_name_index	unsigned short
2	34	_signature_index	unsigned short
2	36	_method_idnum	unsigned short
2	38	_max_stack	unsigned short
2	40	_max_locals	unsigned short
2	42	_size_of_parameters	unsigned short

由表 8.3 可知, JDK 8 的 constMethodOop 在 64 位平台上共占用 44 字节的内存空间(关闭指针压缩功能), 所以字节码指令的内存首地址很显然就在这 44 字节的后面。

但是事实上并不是这样的, 前面在讲解 Java 类字段解析时提到内存对齐, 不仅 Java 类的各个字段会进行内存对齐, C++类也会内存对齐, 并且整个 C++类也需要内存对齐。C++类对齐的其中一条规则就是整个类型实例所占的内存空间必须是类型中宽度最大的字段所占内存的整数倍。由于 constMethodOop 中包含指针, 因此该类实例对象所占的内存空间必须是 8 字节的整数倍, 因此最终 constMethodOop 所占的内存大小应该是 48 字节。

constMethodOop 的大小计算出来之后, 便可以计算字节码指令的内存起始地址, 计算公式很简单:

$$0x00000000113e00458 + 0x30 = 0x00000000113e00488$$

注: 48 对应的十六进制数是 0x30。

计算出字节码指令的起始地址后, 可以使用 HSDB 所提供的 mem 工具查看这个地址处的内容。激动人心的时刻就要来临了!

单击 HSDB 工具栏的 Windows->Command Line 按钮打开 HSDB 的命令行工具, 在里面输入 mem 0x00000000113e00488 2, 输出如下内容:

```
hsdb> mem 0x00000000113e00488 2
```



```
0x00000000113e00488: 0x060436601c1b4eca
0x00000000113e00490: 0x29116800ffb10536
```

上面这个 `mem` 命令的最后加了一个选项 2，表示从 `0x00000000113e00488` 这个内存位置开始，连续输出两行内容，在 64 位平台上，一行显示 128 字节内容。

这一串数字看不懂？很正常。对于不正常的事物，需要分析其规律，然后才能分析出其本质。JVM 的每个字节码指令都占 1 字节长度，并且 `Test.add()` 方法里的局部变量的值也都小于 255，因此每个变量的值也都只占 1 字节的长度，这些变量将作为带参数的字节码指令的操作数，例如 `istore 4`，`istore` 指令后面紧跟的 4，在内存中仅使用 1 字节来表示，并且作为字节码指令的操作数，其内存位置与字节码指令所在的内存相邻，这一点很好理解。

弄清楚这一规律，现在来分析 `mem 0x00000000113e00488 2` 命令的输出结果。输出结果一共 2 行，每行按照从高位到低位的顺序显示，但是字节码指令在内存中的分配顺序实际上是从低位到高位，即 Java 方法的第一条字节码指令所在的内存地址，一定是在最低位，而 Java 方法的最后一条字节码指令，则一定在最高位。所以这里首先要将 `mem` 的输出结果的顺序进行逆转，将其按照从低位到高位顺序重新排列，这样才符合人类的阅读习惯。对于 `Test.add()` 方法，其每个字节码指令及操作数皆使用 1 字节表示，因此在对 `mem` 输出结果的顺序进行逆转时，也以 1 字节为单位进行逆转。内存重排后的内容如下：

```
0x00000000113e00488: ca 4e 1b 1c 60 36 04 06
0x00000000113e00490: 36 05 b1 ff 00 68 11 29
```

内存重排后，面对这一串数字，还是看不懂？别着急，先来看 `Test.add()` 方法所对应的字节码指令的十六进制数字。使用 `javap` 命令分析编译后的 `Test.class` 文件，得到如下结果：

```
public void add(int, int);
Code:
    Stack=2, Locals=6, Args_size=3
    0:   aload_0
    1:   astore_3
    2:   iload_1
    3:   iload_2
    4:   iadd
    5:   istore 4
    7:   iconst_3
    8:   istore 5
   10:   return
```

使用 `javap` 命令只能得到字节码指令的名称，但是内存里实际存储的是字节码的十六进制编码。对照字节码指令集的十六进制编码表（这张编码表就不在本书贴出来了，从网络可以搜索到），将上述字节码指令名称及操作数逐个翻译成十六进制编码，翻译后的结果如下：

```

public void add(int, int);
Code:
  Stack=2, Locals=6, Args_size=3
  0:   aload_0           2a
  1:   astore_3           4e
  2:   iload_1            1b
  3:   iload_2            1c
  4:   iadd                60
  5:   istore 4            36 04
  7:   iconst_3            06
  8:   istore 5            36 05
 10:   return              b1

```

Test.add()方法的第一条字节码指令是 `aload_0`，其十六进制编码是 `2a`，该字节码指令的内存地址就是 `add()`方法的字节码指令在 `constMethodOop` 之后的首地址，因此其内存地址就是上文所分析得到的 `0x0000000113e00488`，也即上面使用 `mem` 命令所观察的内存地址。以 `aload_0` 指令的十六进制编码为开始，将 `add()`方法的字节码指令的十六进制编码以字节为单位顺序编排，如下：

```

0x0000000113e00488: 2a 4e 1b 1c 60 36 04 06
0x0000000113e00490: 36 05 b1

```

将这个结果与上面使用 `mem` 命令所查看到的内存内容进行比较，从第一个字节到 `add()`方法所对应的最后一条字节码指令的十六进制编码 `b1` 逐一比较，会发现除了第一个字节不同之外，其余的都是完全相同的。字节码指令存储的奥秘到此便完全揭晓。

不过仍然有一个疑问，为何第一个字节码的内容不一样呢？这是因为为了使用 HSDB 观察 Test 程序的运行时内存结构，使用了 JDB 进行调试，并且就在 Test.add()方法内部打上了断点，所以 JVM 将 `add()`方法的入口地址改写成断点指令的。等 `add()`方法从中断恢复之后，第一条字节码指令会恢复正常。

其实对于字节码指令的内存位置的计算，除了上面所述的通过 `constMethodOop` 内存首地址加上该对象实例所占的内存大小的方法之外，还有一个更简单的办法，那就是直接查看堆栈（前面章节已经详细讲解过堆栈的结构）。Java 的每一个方法都会在 JVM 的内存中被分配一个栈帧空间，并且栈帧中会保存一个指针指向 Java 方法的字节码指令的内存首地址，而 HSDB 可以观察到运行期的 Java 方法的堆栈内容，因此可以使用这种方式直接定位到 Java 方法的字节码指令所在的内存地址。

使用 JDB 在 Test.add()方法内打断点并调试运行至断点处，接着使用 HSDB 连接 Test 程序。HSDB 刚连上 Test 程序时会显示 JVM 的线程，其中包含主线程，如图 8.8 所示。

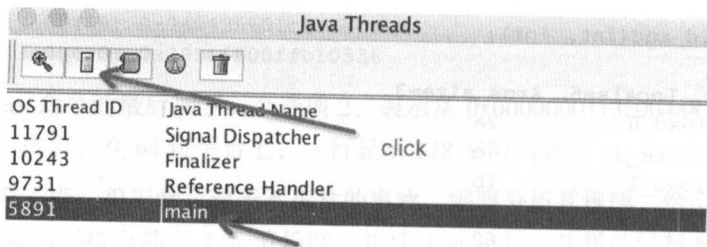


图 8.8 在 HSDB 中显示 Java 进程的主线程

选择主线程 main, 并单击工具栏中的第 2 个按钮, HSDB 便会显示这个主线程的堆栈结构。由于 JDB 在 Test.add() 方法内部打了断点, 并且 Test 程序已经运行至 Test.add() 方法, 因此 Test 主线程的堆栈会包含 add() 方法的栈帧结构。而 add() 方法的栈帧中有一项是一个指针, 该指针指向 add() 方法所对应的 Java 字节码指令的内存地址, 如图 8.9 所示。

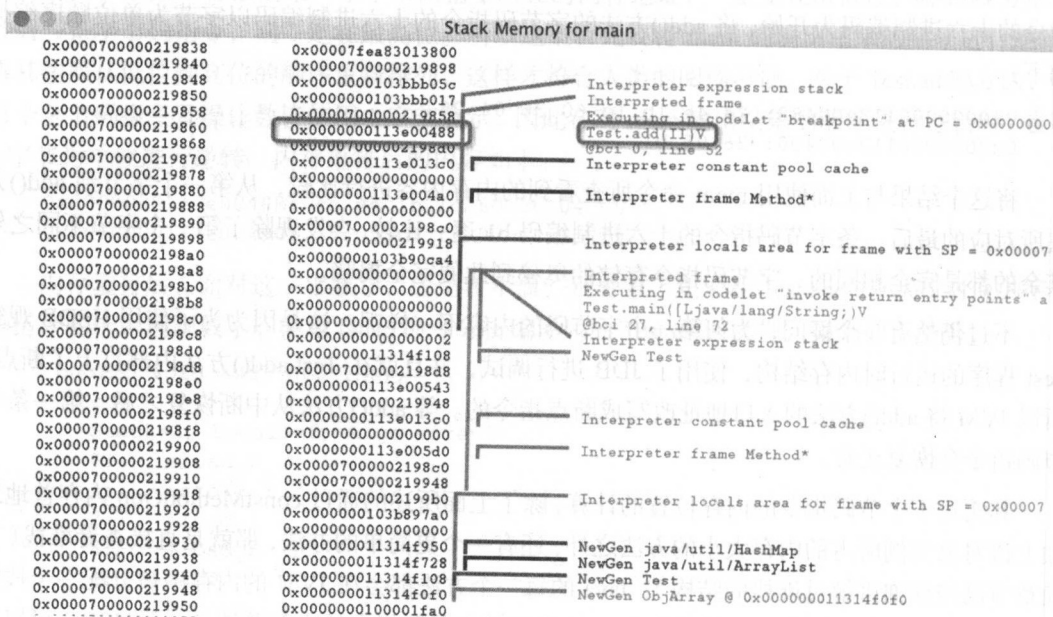


图 8.9 使用 HSDB 查看 Java 方法堆栈

图 8.9 中所框住的内容便是 Test.add() 方法的字节码指令的内存首地址。这个地址与上面所计算出来的地址完全相同。通过本示例也可以猜想 JVM 的垃圾回收机制, 如果一个内存对象没有被栈帧引用或者被栈帧所引用的对象的堆所引用, 则必定是垃圾对象, 则 JVM 的垃圾回收器便可以回收。不过本书限于篇幅, 不会讲解垃圾回收的机制和实现, 将放到下一本书中进行分析。

8.7 vtable

本节开始讲解 Java 多态中的一个十分关键的技术——vtable。

8.7.1 多态

Java 是一门面向对象的编程语言，面向对象的一大特色便是多态。多态的具体体现便是在运行期能够根据对象实例的不同而执行不同的接口方法，换成业界对多态的标准定义便是：允许不同类的对象对同一消息做出响应，即同一消息可以根据发送对象的不同而采用多种不同的行为方式（发送消息就是函数调用）。多态是面向对象编程的特性，而这种特性并不仅仅是喊喊口号就算的，而是必须使用特定的机制或技术去实现。实现多态的技术称为动态绑定（dynamic binding），是指在执行期间判断所引用对象的实际类型，根据其实际的类型调用其相应的方法。在 Java 中，动态绑定也叫“晚绑定”，这是因为在 Java 中还有一类绑定是在编译期间便能确定，所以所谓的晚绑定的概念，是相对于编译期绑定而言的。

面向对象编程语言之所以要实现多态这一特性，最主要的目的就是为了解除类型之间的耦合关系，通俗地讲就是解耦。从计算机软件一产生，“解耦”便是一切计算机程序所要重点考虑的原则之一。其实何止是软件，计算机硬件之间也是以解耦为主要原则的，这类例子举不胜举，例如内存插槽、IO 接口之类，都是实现解耦的手段。

解耦的最大好处在于，一旦系统发生了变化，能够将变化降低到最小，仅变化新增的部件，而对于已经存在的部件，则尽量保持不变。所以一个优秀的系统设计师总是想办法设计拥有良好兼容性和扩展性的架构，而面向对象语言的多态性，则是从语言特性上直接实现对象的解耦，这极大地提升了面向对象编程语言构建一套高内聚、低耦合系统的能力。

由于多态通过“动态绑定”的方式得以实现，而绑定通俗一点讲就是让不同的对象对同一个函数进行调用，或者反过来讲就是将同一个函数与不同的对象绑定起来，所以多态性得以实现的一个大前提就是，编程语言必须是面向对象的，否则哪来的函数与对象相互绑定一说呢？同时，函数与对象相互绑定，意味着函数也属于对象的一部分，这便具备了封装的特性。因为有了封装，才有了对象。有了对象才叫作面向对象编程。同时，一个函数能够绑定多个不同的对象，意味着多个不同的对象都具有相同的行为，这是继承的含义。因此，面向对象编程语言的三大特性——封装、继承与多态，其中前两个特性“封装”与“继承”其实就是为了第三个特性“多态”而准备的，或者说“封装”与“继承”成全了“多态”，为“多态”做了嫁衣。

下面是一个简单的动态绑定的示例程序：

清单：/Test.java

作用：演示动态绑定

```
public abstract class Animal {
    abstract void say();

    public static void main(String[] args){
        Animal animal = new Dog();
        run(animal);

        animal = new Cat();
        run(animal);
    }

    public static void run(Animal animal){
        animal.say();
    }
}

class Dog extends Animal{
    @Override
    void say() {
        System.out.println("I'm a dog");
    }
}

class Cat extends Animal{
    @Override
    void say() {
        System.out.println("I'm a cat");
    }
}
```

本示例程序中定义了虚类 `Animal`，同时定义了 2 个子类 `Dog` 和 `Cat`，这 2 个子类都重写了基类中的 `say()` 方法。在 `main()` 函数中，将 `animal` 实例引用分别指向 `Dog` 和 `Cat` 的实例，并分别调用 `run(Animal)` 方法。在本示例中，当在 `Animal.run(Animal)` 方法中执行 `animal.say()` 时，因为在编译期并不知道 `animal` 这个引用到底指向哪个实例对象，所以编译期无法进行绑定，必须等到运行期才能确切知道最终调用哪个子类的 `say()` 方法，这便是动态绑定，也即晚绑定，这是 Java 语言以及绝大多数面向对象语言的动态机制最直接的体现。

8.7.2 C++中的多态与 vtable

JVM 实现晚绑定的机制基于 vtable，即 virtual table，也即虚方法表。JVM 通过虚方法表在运行期动态确定所调用的目标类的目标方法。在讲解 JVM 的 vtable 概念之前，先一起品味 C++ 中虚方法表的实现机制，这两者有很紧密的联系。

有如下 C++ 类：

清单：/Test.cpp

作用：演示 C++ 的动态绑定

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

class CPLUS{
public:
    short x;
public:
    void run(){
        this->x=2;
    }
};

int main(int argc, char const *argv[])
{
    CPLUS cplus;
    printf("sizeof(CPLUS)= %lu\n", sizeof(CPLUS));
    printf("&cplus= %p\n", &cplus);
    printf("&(cplus.x)= %p\n", &(cplus.x));

    return 0;
}
```

这个 C++ 示例很简单，类中包含一个 short 类型的变量和一个 run() 方法，在 main() 函数中打印 3 个信息：CPLUS 类型宽度、其实例的内存首地址和其变量 x 的内存地址。编译并执行，输出结果如下：

```
sizeof(CPLUS)= 2
&cplus= 0x7fff5ef57998
&(cplus.x)= 0x7fff5ef57998
```

由于 CPLUS 类中仅包含 1 个 short 类型的变量，因此该类型的数据宽度自然是 2。另外注意观察结果中的 cplus 实例和其变量 x 的内存地址，两者是相等的。

现在将 C++ 类中的 run() 方法修改一下，变成虚方法，修改后的程序如下：

清单：/Test.cpp

作用：演示 C++ 的动态绑定

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

class CPLUS{
public:
    short x;
public:
    virtual void run(){
        this->x=2;
    }
};
```

main()函数内容不变，因此这里不重复贴出来。编译并运行程序，现在输出变成如下所示：

```
sizeof(CPLUS)= 16
&cplus= 0x7fff5694d990
&(cplus.x)= 0x7fff5694d998
```

注意看，现在 sizeof(CPLUS)的值变成 16 了，并且 cplus 实例和其变量 x 的内存地址也不再相等了。这是咋回事呢？这是因为当 C++ 类中出现虚方法时，表示该方法拥有多态性，此时会根据类型指针所指向的实际对象而在运行期调用不同的方法，这与 Java 中的多态在语义上是完全一致的。

C++ 为了实现多态，就在 C++ 类实例对象中嵌入虚函数表 vtable，通过虚函数表来实现运行期的方法分派。C++ 中所谓虚函数表，其实就是一个普通的表，表中存储的是方法指针，方法指针会指向目标方法的内存地址。所以虚函数表就是一堆指针的集合而已。

对于大部分 C++ 编译器而言，其实现虚函数表的机制都大同小异，都会将虚函数表分配在 C++ 对象实例的起始位置，当 C++ 类中出现虚函数表时，其内存分配就是先分配虚函数表，再分配类中的字段空间。以本示例程序而言，CPLUS 的实例对象 cplus 的实际内存结构如图 8.10 所示。

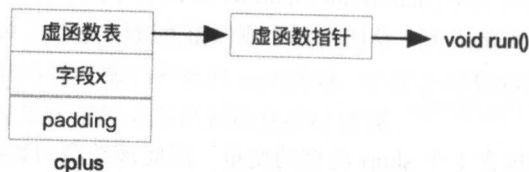


图 8.10 cplus 实例内存结构布局

由于 CPLUS 类中仅包含一个虚函数，因此虚函数表中只有一个指针。

注意，在 `cplus` 实例的末尾有一段补白空间，这是因为 C++ 编译器会对类型做对齐处理，整个 C++ 类实例对象所占的内存空间必须是其中宽度最大的字段所占内存空间的整数倍，而 CPLUS 类中由于嵌入了虚函数表，表中元素是指针类型，在 64 位平台上，1 个指针占 8 字节内存空间，因此 CPLUS 类实例对象所占的内存空间就是 16 字节，这就是上面运行示例程序后输出结果中的 `sizeof(CPLUS)` 的值变成 16 的原因所在。同时，字段 `x` 被安排在虚函数表之后，因此 `x` 的内存地址也不再与 `cplus` 实例对象的内存首地址相等，并且根据上述程序运行结果可见，这两者的内存地址相差 8 字节，这正好是一个指针的宽度。

8.7.3 Java 中的多态实现机制

Java 的多态机制并没有跳出这个圈，也采用了 `vtable` 来实现动态绑定。Java 类在 JVM 内部对应的对象是 `instanceKlassOop` (JDK 8 中是 `instanceKlass`)。在 JVM 加载 Java 类的过程中，JVM 会动态解析 Java 类的方法及其对父类方法的重写，进而构建出一个 `vtable`，并将 `vtable` 分配到 `instanceKlassOop` 内存区的末尾，从而支持运行期的方法动态绑定。

JVM 的 `vtable` 机制与 C++ 的 `vtable` 机制之间最大之不同在于，C++ 的 `vtable` 在编译期间便由编译器完成分析和模型构建，而 JVM 的 `vtable` 则在 JVM 运行期、Java 类被加载时进行动态构建。其实也可以认为 JVM 在运行期做了 C++ 编译器在静态编译期所做的事情。关于 C++ 编译器在编译期间构建 `vtable` 的机制，不同的编译器具体的实现方式不尽相同，但基本都没跳出这个框框。各位有兴趣的道友如果想对此进行研究，可以分析 C++ 类编译后的汇编脚本。

JVM 在第一次加载 Java 类时会调用 `classFileParser.cpp::parseClassFile()` 函数对 Java class 字节码进行解析，上文刚刚讲过，在 `parseClassFile()` 函数中会调用 `parse_methods()` 函数解析 Java 类中的方法，`parse_methods()` 函数执行完之后，会继续调用 `klassVtable::compute_vtable_size_and_num_mirandas()` 函数，计算当前 Java 类的 `vtable` 大小。下面便一起来围观该方法实现的主要逻辑：

清单：/src/share/vm/oops/klassVtable.cpp

作用：verify_legal_method_name()方法主要逻辑

```
void klassVtable::compute_vtable_size_and_num_mirandas(int &vtable_length,
                                                         int &num_miranda_methods,
                                                         klassOop super,
                                                         objArrayOop methods,
                                                         AccessFlags class_flags,
                                                         Handle classloader,
                                                         Symbol* classname,
```

```

                                objArrayOop local_interfaces,
                                TRAPS
                                ) {
    vtable_length = 0;
    num_miranda_methods = 0;

    instanceKlass* sk = (instanceKlass*)super->klass_part();
    vtable_length = super == NULL ? 0 : sk->vtable_length();

    int len = methods->length();
    for (int i = 0; i < len; i++) {
        methodHandle mh(THREAD, methodOop(methods->obj_at(i)));

        if (needs_new_vtable_entry(mh, super, classloader, classname,
            class_flags, THREAD)) {
            vtable_length += vtableEntry::size(); // we need a new entry
        }
    }

    num_miranda_methods = get_num_mirandas(super, methods, local_interfaces);
    vtable_length += (num_miranda_methods * vtableEntry::size());

    //...
}

```

从这段代码可以看出计算 vtable 的思路主要分为两步：

- (1) 获取父类 vtable 的大小，并将当前类的 vtable 的大小设置为父类 vtable 的大小。
- (2) 循环遍历当前 Java 类的每一个方法，调用 needs_new_vtable_entry() 函数进行判断，如果判断的结果是 true，则将 vtable 的大小增 1。

关于父类 vtable 的大小需要从 Java 类的顶级父类 java.lang.Object 开始算起，这个一会儿再讲，现在重点看第 2 步——needs_new_vtable_entry() 函数的实现逻辑：

清单：/src/share/vm/oops/klassVtable.cpp

作用：needs_new_vtable_entry() 方法主要逻辑

```

bool klassVtable::needs_new_vtable_entry(methodHandle target_method,
                                           klassOop super,
                                           Handle classloader,
                                           Symbol* classname,
                                           AccessFlags class_flags,
                                           TRAPS) {
    //如果 Java 方法被 final、static 修饰，或者 Java 类被 final 修饰，或者 Java 方法是构造
    函数，则返回 false
    if ((class_flags.is_final() || target_method()->is_final()) ||
        (target_method()->is_static()) ||

```

```

    (target_method()->name() == vmSymbols::object_initializer_name())
    ) {
        return false;
    }
    //...
    //如果 Java 方法的访问权限是 private 则返回 true
    if (target_method()->is_private()) {
        return true;
    }

```

//遍历父类中同名、签名也完全相同的方法，如果父类方法的访问权限是 public 或者 protected，并且没有 static 或 private 修饰，则说明子类重写了父类的方法，此时返回 false

```

ResourceMark rm;
Symbol* name = target_method()->name();
Symbol* signature = target_method()->signature();
klassOop k = super;
methodOop super_method = NULL;
instanceKlass *holder = NULL;
methodOop recheck_method = NULL;
while (k != NULL) {
    super_method = instanceKlass::cast(k)->lookup_method(name, signature);
    if (super_method == NULL) {
        break;
    }

    instanceKlass* superk = instanceKlass::cast(super_method->method_holder());
    if ((!super_method->is_static()) &&
        (!super_method->is_private())) {
        if (superk->is_override(super_method, classloader, classname, THREAD)) {
            return false;
        }
    }
    k = superk->super();
}

//处理 miranda 方法
instanceKlass *sk = instanceKlass::cast(super);
if (sk->has_miranda_methods()) {
    if (sk->lookup_method_in_all_interfaces(name, signature) != NULL) {
        return false;
    }
}
return true;
}

```

在分析这段逻辑之前，有必要稍微解释一下 vtable 的机制。Java 类在运行期进行动态绑定

的方法,一定会被声明为 `public` 或者 `protected` 的,并且没有 `static` 和 `final` 修饰,且 Java 类上也没有 `final` 修饰。道理很简单,阐述如下:

- ◎ 如果一个 Java 方法被 `static` 修饰,则压根儿不会参与到整个 Java 类的继承体系中,所以静态的 Java 方法不会参与到运行期的动态绑定机制。所谓的动态绑定,是指将 Java 类实例与 Java 方法搭配,而静态方法的调用压根儿不需要经过类实例,只需要有类型名即可。
- ◎ 如果一个 Java 方法被 `private` 修饰,则外部根本无法调用该方法,只能被该类内部的其他方法所调用,因此也不会参与到运行期的动态绑定。
- ◎ 如果一个 Java 方法被 `final` 修饰,则其子类无法重写该方法(如果非要重写,则 IDE 会报错的,并且编译也不会通过)。既然子类无法重写该方法,则该方法仅为 Java 类所固有,不会出现多态性。
- ◎ 如果一个 Java 类被 `final` 修饰,则该 Java 类中的所有非静态方法都会隐式地被 `final` 修饰,参考第 3 条,则该 Java 类中的所有非静态方法都不会被子类重写,因此都不会出现多态性,不会发生运行期动态绑定。

只有满足以上 4 个条件的 Java 方法,才有可能参与到运行期的动态绑定,而满足了以上 4 个条件的 Java 方法,其一定只被 `public` 或者 `protected` 关键字修饰。而满足了这 4 个条件的 Java 方法只是有可能参与动态绑定,这是因为仅仅满足了这 4 个条件还不够,还得有别的条件,其余的条件包括以下:

- ◎ 父类中必须包含名称相同的 Java 方法。这里所谓的父类,并不一定是 Java 类的直接父类,也可能是间接的父类。例如 A 类继承了 B 类, B 类继承了 C 类,则 A 类间接继承了 C 类。
- ◎ 父类中名称相同的 Java 类,其签名也必须完全一致。

以上这两点其实换而言之,就是 Java 类中的方法必须重写了父类的 Java 方法,这样的 Java 类方法才会参与到运行期动态绑定。而这其实正是多态的含义:父类与子类中包含一个完全一样的行为(即 Java 方法的名称和签名完全相同),在运行期这一行为将根据不同的条件与具体的类型对象相绑定(父类或子类实例)。

而父类与子类都同时拥有的相同的行为,从继承的角度看,就是子类对父类的重写,并且重写的前提是,这一行为不能是 `private` 的,不能是 `static` 的,不能是 `final` 的,否则父类的行为无法被子类继承,所谓的重写更无从谈起了。所以上面这段代码的逻辑就是在进行这一系列的判断,只有最终满足条件的,才会返回 `true`。

在 `klassVtable.cpp::compute_vtable_size_and_num_mirandas()` 函数中, 如果 `needs_new_vtable_entry()` 函数返回 `true`, 将会对 `vtable_length` 增加 1。

现在我们描述 JVM 内部 `vtable` 的实现机制。每一个 Java 类在 JVM 内部都有一个对应的 `instanceClassOop`, `vtable` 就被分配在这个 `oop` 内存区域的后面。`vtable` 表中的每一个位置存放一个指针, 指向 Java 方法在内存中所对应的 `methodOop` 的内存首地址。如果一个 Java 类继承了父类, 则该 Java 类就会直接继承父类的 `vtable`。若该 Java 类中声明了一个非 `private`、非 `final`、非 `static` 的方法, 若该方法是对父类方法的重写, 则 JVM 会更新父类 `vtable` 表中指向父类被重写的方法的指针, 使其指向子类中该方法的内存地址。若该方法并不是对父类方法的重写, 则 JVM 会向该 Java 类的 `vtable` 中插入一个新的指针元素, 使其指向该方法的内存位置。

相信很多人对这段文字看得有些晕, 还是举个例子。假设有下面一个超类:

清单: /A.java

作用: Java 的多态机制

```
class A {
    public void print(){
        System.out.println("A.print()");
    };
}
```

接着定义一个子类:

清单: /B.java

作用: Java 的多态机制

```
class B extends A{
    public void newFun(){
        System.out.println("B.newFun()");
    }

    public void print(){
        System.out.println("B.print()");
    }
}
```

子类 B 继承于类 A, 并且重写了类 A 的 `void print()` 方法。由于类 B 和类 A 中的 `print()` 方法名称相同, 签名也完全相同, 并且都没有 `private`、`static`、`final` 这 3 个关键字修饰, 因此该方法将会在运行期进行动态绑定。

当 HotSpot 在运行期加载类 A 时, 其 `vtable` 中将会有一个指针元素指向其 `void print()` 方法在 HotSpot 内部的内存首地址。当 HotSpot 加载类 B 时, 首先类 B 完全继承其父类 A 的 `vtable`, 因此类 B 便也有一个 `vtable`, 并且 `vtable` 里有一个指针指向类 A 的 `print()` 方法的内存地址。

HotSpot 遍历类 B 的所有方法，并发现 print()方法是 public 的，并且没有被 static、final 修饰，于是 HotSpot 去搜索其父类中名称相同、签名也相同的方法（即上文所讲的 klassVtable.cpp::needs_new_vtable_entry()函数），结果发现父类中存在一个完全一样的方法，于是 HotSpot 就会将类 B 的 vtable 中原本指向类 A 的 print()方法的内存地址的指针值修改成指向类 B 自己的 print()方法所在的内存地址。

而当 HotSpot 解析类 B 的 newFun()方法时，由于该方法并没有在父类中出现，并且也是 public 的，同时没有 static 和 final 修饰，满足 vtable 的条件，于是 HotSpot 将类 B 原本继承于 A 的 vtable 的长度增 1，并将新增的 vtable 的指针元素指向 newFun()方法在内存中的位置。

在 classFileParser.cpp::parseClassFile()函数中，执行完 klassVtable.cpp::compute_vtable_size_and_num_mirandas()函数后，会得到当前 Java 类的 vtable 的大小，虚函数表的大小被保存在 classFileParser.cpp::parseClassFile()函数的局部变量 vtable_size 中，该变量值将在后续创建 Java 类所对应的 instanceClassOop 对象时被保存到该对象中的 _vtable_len 字段中，而该字段可以通过 HSDB 进行查看，并进而验证 HotSpot 计算虚函数表大小的逻辑。

写个测试类调用上述示例中的类 A 或 B 中的方法并设断点，让程序运行到断点处中断，然后使用 HSDB 连上测试程序（由于测试程序很简单，因此这里就不贴出来），单击 HSDB 工具栏上的 Tools->Class Browser 按钮，会看到类 A 和类 B，如图 8-11 所示。

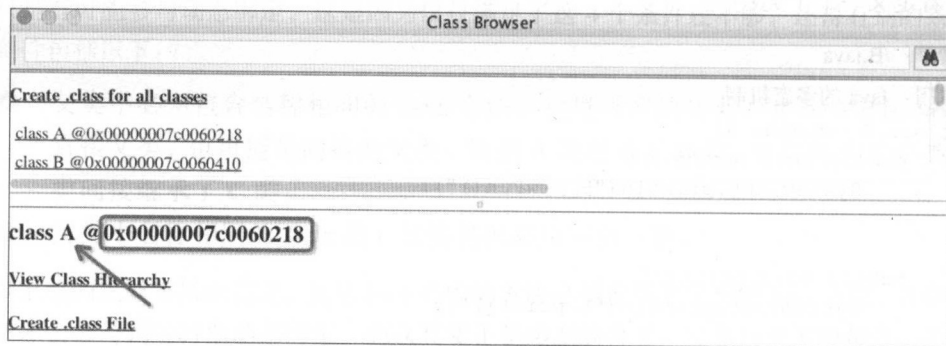


图 8.11 使用 HSDB 查看类 A 的内存地址

复制图 8.11 中的类 A 的地址 0x00000007c0060218，单击 HSDB 工具栏的 Tools->Inspector 按钮，将该地址复制进去，可以查看类 A 在 JVM 内部所对应的 instanceClassOop 的内部结构，其中就有类 A 所对应的 vtable 虚函数表的大小，如图 8.12 所示。

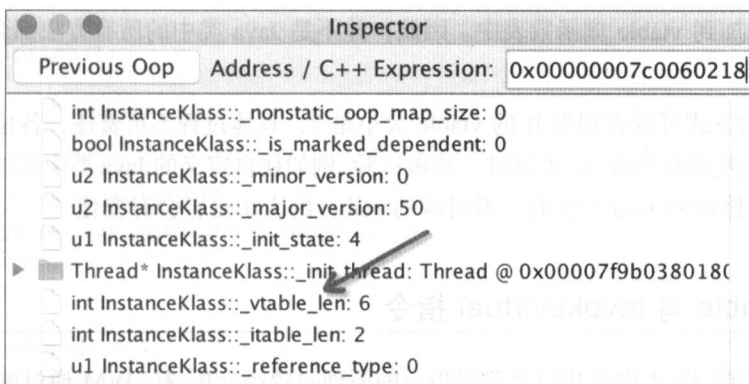


图 8.12 使用 HSDB 查看类 A 所对应的 instanceClassOop 的 vtable 大小

如图 8.12 中箭头所示,类 A 的 vtable 大小为 6。这里有个疑问,类 A 中只定义了一个方法,按理说其 vtable 大小应该是 1,可是为何 HSDB 里却显示 6 呢? 其实,在 Java 语言中,所有 Java 类都隐式继承了顶级父类 `java.lang.Object`,类 A 的 vtable 的另外 5 个方法指针元素便指向 `java.lang.Object` 中的 5 个方法。`java.lang.Object` 中一共定义了如下 12 个方法:

```
protected void finalize()
public final void wait(long, int)
public final native void wait(long)
public final void wait()
public boolean equals(java.lang.Object)
public java.lang.String toString()
public native int hashCode()
public final native java.lang.Class getClass() [signature ()Ljava.lang.
Class<*>;]
protected native java.lang.Object clone()
private static native void registerNatives()
public final native void notify()
public final native void notifyAll()
```

而其中只有如下 5 个方法可以被子类重写,因此 `java.lang.Object` 类的 vtable 大小为 5,这 5 个方法如下:

```
protected void finalize()
public boolean equals(java.lang.Object)
public java.lang.String toString()
public native int hashCode()
protected native java.lang.Object clone()
```

这 5 个方法都是 `public` 或者 `protected` 的,并且没有用 `static` 和 `final` 修饰,而 `java.lang.Object` 中的其他方法要么被 `final` 修饰,要么被 `static` 修饰,因此都不能被子类继承,所以其方法指针

不会被 JVM 添加到 vtable 虚函数表中。因此，并不是 Java 类中的所有方法都会被放入 vtable 中。

使用同样的方式可以看到类 B 的 vtable 大小是 7，具体过程不再赘述，各位道友可自行试验分析。同时，使用同样的办法，可以进一步做试验，例如往自定义的 Java 类中添加使用 private、static 或者 final 修饰的 Java 方法时，看对应的 vtable 的大小是否会有变化。

8.7.4 vtable 与 invokevirtual 指令

本书一开始讲 JVM 内部执行字节码指令的原理时曾经分析过，JVM 通过调用 CallStub 例程开始调用 Java 程序的主函数 main()，在 CallStub 例程内部最终调用了 zero_locals 这个例程，而在 zero_locals 例程中最终跳转到 Java 程序的 main() 主函数的第一条字节码指令并开始执行 Java 程序。那么在 Java 程序内部，当一个 Java 方法调用了另一个 Java 方法时，是如何实现 Java 方法的调用的？这得从 Java 的字节码指令开始说起。

Java 的字节码指令中方法的调用实现分为 4 种指令：

- ◎ Invokevirtual，为最常见的情况，包含 virtual dispatch（虚方法分发）机制。
- ◎ Invokespecial，调用 private 和构造方法，绕过了 virtual dispatch。
- ◎ Invokeinterface，其实现与 invokevirtual 类似。
- ◎ Invokestatic，调用静态方法。

其中最复杂的要属 invokevirtual 指令，它涉及多态的特性，凡是 Java 类中需要在运行期动态绑定的方法调用，都通过 invokevirtual 指令，该指令将实现方法的分发，因此 vtable 与该指令之间有莫大的联系，而事实上，在 HotSpot 执行 invokevirtual 指令的过程中，最终会读取被调用的类的 vtable 虚函数表，并据此决定真实的目标调用方法。JVM 内部实现 virtual dispatch 机制时，会首先从 receiver（被调用方法的对象）的类的实现中查找对应的方法，如果没找到，则去父类查找，直到找到函数并实现调用，而不是依赖于引用的类型。这里还是先来看一个例子，感受一下上面这 4 个指令的用法：

清单：/Test.java

作用：vtable 与 invokevirtual 字节码指令验证

```
public class Test {
    public static void main(String[] args) {
        A a = new A();
        a.print();

        B b = new B();
```

```

        b.newFun();
        b.print();
    }
}

class A {
    public void print(){
        System.out.println("A.print()");
    };
}

class B extends A{
    public void newFun(){
        System.out.println("B.newFun()");
    }

    public void print(){
        System.out.println("B.print()");
        newFun();
        privateFun();
    }

    private void privateFun(){ }
}

```

编译这段程序,并使用 `javap` 命令分别分析 `Test.main()` 函数中的 3 个方法调用的字节码指令,以及类 `B.print()` 方法中调用 `newFun()` 和 `privateFun()` 时所使用的字节码指令。

首先看 `B` 类的 `print()` 方法调用 `newFun()` 和 `privateFun()` 方法时的字节码指令, `javap` 分析的结果如下所示:

```

public void print();
Code:
Stack=2, Locals=1, Args_size=1
0:  getstatic      #2; //Field java/lang/System.out:Ljava/io/PrintStream;
3:  ldc            #5; //String B.print()
5:  invokevirtual  #4; //Method
java/io/PrintStream.println:(Ljava/lang/String;)V
8:  aload_0
9:  invokevirtual  #6; //Method newFun:()V
12: aload_0
13: invokespecial  #7; //Method privateFun:()V
16:  return

```

由 `javap` 分析的结果可知,在 `B.print()` 方法中调用 `newFun()` 和 `privateFun()` 方法时所使用的字节码指令分别是 `invokevirtual` 和 `invokespecial`,为何在类 `B` 内部调用自己的这两个方法,所

使用的指令竟然还有所不同呢？

这是因为 `newFun()` 是 `public` 的，并且没有 `static` 和 `final` 修饰，因此这个方法是可以被继承的，并且是可以被子类重写的。而编译器在编译期间并不知道类 `B` 有没有子类，因此这里只能使用 `invokevirtual` 指令去调用 `newFun()` 方法，从而使 `newFun()` 方法支持在运行期进行动态绑定。虽然编译器在编译期间可以分析整个工程以确定类 `B` 到底有无子类，但是别忘了 JVM 可是能够支持在运行期动态创建新的类型（例如，使用 `cglib`）的，编译器根本无法得知在运行期会不会突然冒出个类去继承类 `B` 并重写类 `B` 的 `newFun()` 方法。

而 `privateFun()` 方法则不一样，其为类 `B` 的私有方法，就算有子类继承于类 `B`，也无法重写该方法，因此该方法不需要参与动态绑定，在编译期间便能直接确定其调用者，所以其对应的字节码指令是 `invokespecial`。

与 `B.print()` 方法中调用 `newFun()` 方法使用 `invokevirtual` 指令同样的道理，在 `Test` 类的 `main()` 主函数中调用 `a.print()`、`b.print()` 和 `b.newFun()` 这 3 个方法时，所对应的字节码指令也都是 `invokevirtual`，这是因为这 3 个方法都是可以被子类所重写的，所以编译器在编译期间无法确定其真实的调用方到底是谁，只能通过 `invokevirtual` 指令在运行期进行动态绑定。

8.7.5 HSDB 查看运行时 vtable

在 HotSpot 中，Java 类在 JVM 内部所对应的类型是 `instanceKlassOop`（JDK 8 中的类型名是 `instanceKlass`），`vtable` 便分配在 `instanceKlass` 对象实例的内存末尾。`instanceKlass` 对象实例在内存中所占内存大小是 `0x1b8` 字节，换算成十进制是 440。根据这个特点，可以使用 HSDB 获取到 Java 类所对应的 `instanceKlass` 在内存中的首地址，然后加上 `0x1b8`，就得到 `vtable` 的内存地址，如此便可以查看这个内存位置上的 `vtable` 成员数据。

还是以上一节中的 `class A` 作为示例程序，类 `A` 中仅包含 1 个 Java 方法，因此类 `A` 的 `vtable` 长度一共是 6，另外 5 个是超类 `java.lang.Object` 中的 5 个方法。使用 JDB 基于 JDK 8 调试（关闭 JDK 的指针压缩选项），并运行至断点处使程序暂停，然后使用 HSDB 连接上测试程序，打开 HSDB 的 `Tools->Class Browser` 功能，就能看到类 `A` 在 JVM 内部所对应的 `instanceKlass` 对象实例的内存地址，如图 8.13 所示。

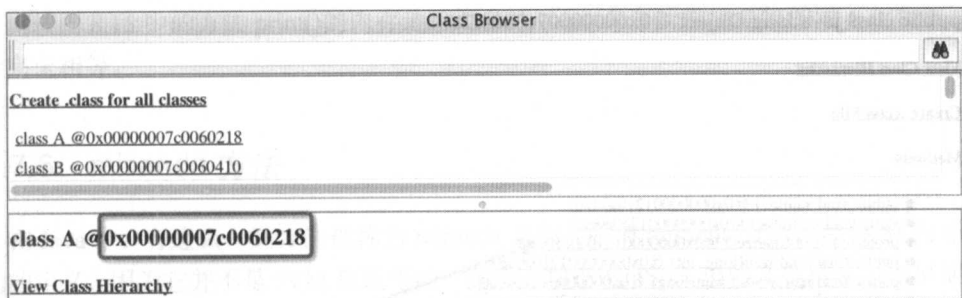


图 8.13 使用 HSDB 查看类 A 的内存地址

由图 8.13 可知,类 A 在 JVM 内部所对应的 instanceClass 的内存首地址是 0x00000007c0060218, 由于 vtable 被分配在 instanceClass 的末尾位置, 因此 vtable 的内存首地址是:

$$0x00000007c0060218 + 0x1b8 = 0x00000007c00603d0$$

这里的 0x1b8 是 instanceClass 对象实例所占的内存空间大小。得到 vtable 内存地址后, 便可以使用 HSDB 的 mem 工具来查看这个地址处的内存数据。单击 HSDB 工具栏上的 Windows->Console 按钮, 打开 HSDB 的终端控制台, 按回车键, 然后输入“mem 0x00000007c00603d0 6”命令, 就可以查看从 vtable 内存首地址开始的连续 6 个双字内容, 如下所示:

```
hsdb> mem 0x00000007c00603d0 6
0x00000007c00603d0: 0x000000001210a0c10
0x00000007c00603d8: 0x000000001210a06e8
0x00000007c00603e0: 0x000000001210a0840
0x00000007c00603e8: 0x000000001210a0640
0x00000007c00603f0: 0x000000001210a0778
0x00000007c00603f8: 0x000000001214a05d8
```

在 64 位平台上, 一个指针占 8 字节, 而 vtable 里的每一个成员元素都是一个指针, 因此这里 mem 所输出的 6 行, 正好是类 A 的 vtable 里的 6 个方法指针, 每一个指针指向 1 个方法在内存中的位置。类 A 的 vtable 总长度是 6, 其中前面 5 个是基类 java.lang.Object 中的 5 个方法的指针。在 HSDB 中可以查看 Java 方法在 JVM 内部所对应的 method 对象实例的内存地址, 单击 HSDB 工具栏上的 Tools->Class Browser 按钮, 选择某个类, HSDB 会显示这个类中的所有方法及其内存地址。图 8.14 显示了使用 HSDB 查看 java.lang.Object 中的方法的内存地址。

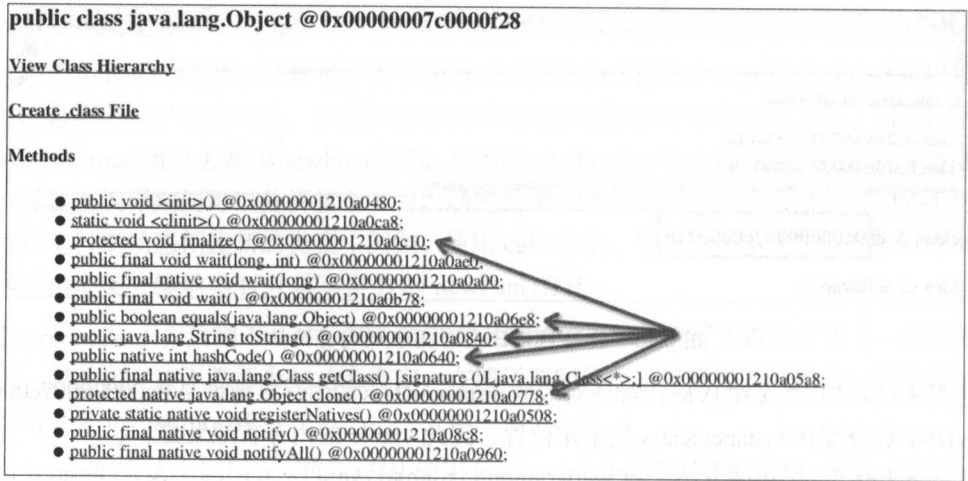


图 8.14 使用 HSDB 查看 java.lang.Object 中的 5 个动态方法的内存地址

图 8.14 标识出了 java.lang.Object 中 5 个可被继承的 Java 方法的内存地址。既然类 A 的 vtable 的 5 个成员指针指向 java.lang.Object 中的这 5 个方法，则 vtable 中的前 5 个指针的值必定就是 java.lang.Object 类中这 5 个方法的内存地址。比较上面 mem 命令的输出结果与该图中的这 5 个地址的值，果然发现 mem 命令所输出结果的前 5 行与 java.lang.Object 中的这 5 个方法的内存地址是一一对应的。这一方面证明 vtable 数据的确是被分配在 instanceKlass 对象实例的内存区域的后面，另一方面也说明 vtable 中所存储的的确是指向方法内存的指针。

上面 mem 命令所输出的第 6 行的指针，一定就是指向类 A 自己的方法的内存地址。使用 HSDB 查看类 A 的方法的内存地址，如图 8.15 所示。

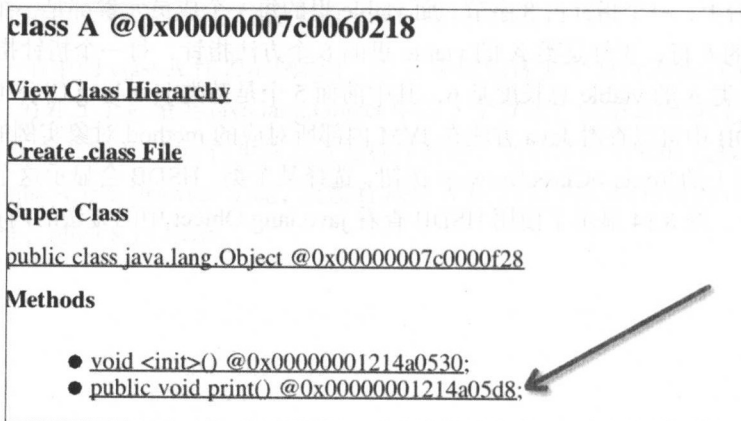


图 8.15 使用 HSDB 查看类 A 的 print() 方法的内存地址

比较图 8.15 中的方法 `print()` 的内存地址与上述 `mem` 命令所输出结果的第 6 行数据, 会发现两者完全相等。

8.7.6 miranda 方法

在 Java 中, 有这么一类方法被称为 miranda 方法, 所谓 miranda 方法, 在 JVM 内部并没有专门的定义, 因为它并不是 JVM 规范里的一部分。根据 HotSpot 里的文档描述, 在早期的虚拟机里有一个 bug, 那就是 JVM 在遍历解析 Java 类的方法时, 仅仅会遍历 Java 类及其所有父类的方法, 但是并不会去查找 Java 类所实现的接口 `interface` 里的方法, 这会导致这样一种结果: 如果 Java 类没有实现接口里的方法, 则接口里的方法将不会被虚拟机查到。为了解决这个问题, 编译器引入了一个相反的办法, 那就是在编译期往 Java 类中插入接口里所定义的方法, 这些方法就是所谓的 miranda 方法。但是这种解决办法也是有问题的, 因为 miranda 方法并不是 Java 规范的一部分, 所以从某种意义上说, 这其实是另一种 bug。

按照上面所描述的, 首先有一个疑问: JVM 规定, 对于接口里所定义的接口方法, Java 类必须全部实现这些接口类, 那么哪里还会出现什么 bug 呢? 但是有一种情况允许 Java 类可以不实现接口方法, 只需要在类名前面加上 `abstract` 修饰符, 如下例所示。

先定义一个接口:

清单: /IA.java

作用: miranda 方法

```
public interface IA {
    void test();
}
```

接着定义一个 `abstract` 的类:

清单: /MyClass.java

作用: miranda 方法

```
public abstract class MyClass implements IA{
    public MyClass(){
        test();
    }
}
```

`MyClass` 类继承了 `IA` 接口, 但是并没有实现接口方法 `test()`, 这种情况也是能够编译通过的。但是在 `MyClass` 的构造函数里, 可以调用 `test()` 这个接口方法。如果编译器没有 miranda 机制, 则在 `MyClass` 类的构造函数里调用 `test()` 接口方法时, 肯定会编译报错, 因为这个方法只存在于接口类中, 而不存在于 `MyClass` 的任何父类中。

事实上，miranda 方法中的“Miranda”一词是有典故的，这个典故来源于法庭宣判。miranda 其实是一个法律术语，即“米兰达原则”，简单来说，米兰达原则（Miranda Rule）要求警察告诉被拘捕的犯罪嫌疑人，他可以对警察保持缄默，他有权要求有律师在场。

关于这个典故各位道友可以自行在网络上搜索，本书不多讲了。总之，JVM 借鉴了法律上的这一充满人道主义关怀的原则，将其运用于接口类的方法实现中。如果一个 Java 类无法提供接口类方法的实现，那么编译器将会为其提供一个方法实现，这个方法就叫作 miranda 方法。这如同法律审判一样，如果一个犯罪嫌疑人没有能力请一名律师辩护，那么法庭将为其提供一个。从这个角度来重新审视程序，会发现程序设计原来和生活法则都是相通的，创意来源于生活，设计来源于生活，一切都离不开生活。

软件程序使用符号作为程序世界的规则，但是依然存在不遵循规则的现象，在程序的世界里，当出现不遵循规则的现象时，设计者总是倾向于来弥补缺陷，使程序的世界尽量完美，少一些“悲剧”，这何尝不是每一个人所梦想并为之奋斗的社会愿景？

在 Java 类中调用 miranda 方法时，实现的是动态绑定策略，而非早绑定。还是以上面的 MyClass 类为例，在其构造函数中调用 test()方法时，生成的汇编指令是 invokevirtual，如下：

```
public MyClass();
Code:
    Stack=1, Locals=1, Args_size=1
    0:   aload_0
    1:   invokespecial    #1; //Method java/lang/Object."<init>":()V
    4:   aload_0
    5:   invokevirtual     #2; //Method test1:()V
    8:   return
```

既然 miranda 方法被实现为“晚绑定”机制，那么按照上面所讲的 vtable 的原理，miranda 方法将会被加入到 vtable 中，而 HotSpot 内部的确也是这么处理的。在 classFileParser.cpp::parseClassFile()函数中调用 klassVtable.cpp::compute_vtable_size_and_num_mirandas()函数计算 vtable 长度时，其实已经包含了对 miranda 方法的处理，如下：

清单：/src/share/vm/oops/klassVtable.cpp

作用：在 compute_vtable_size_and_num_mirandas()函数中处理 miranda 方法

```
void klassVtable::compute_vtable_size_and_num_mirandas(int &vtable_length,
                                                         int &num_miranda_methods,
                                                         klassOop super,
                                                         objArrayOop methods,
                                                         AccessFlags class_flags,
                                                         Handle classloader,
                                                         Symbol* classname,
                                                         objArrayOop local_interfaces,
```



```

        TRAPS
    ) {

//...

    // compute the number of mirandas methods that must be added to the end
    num_miranda_methods = get_num_mirandas(super, methods, local_interfaces);
    vtable_length += (num_miranda_methods * vtableEntry::size());

//...
}

```

在这个逻辑中，HotSpot 先计算出当前类的所有 miranda 方法的总数，并将其增加到 vtable 的长度变量里。而 miranda 方法的总数的计算逻辑就是搜索当前类所实现的所有接口类，以及各个接口所继承的父类接口类中的方法（别忘了接口类是可以继承接口类的），如果这些接口类方法并没有在当前类中实现，则会被当作 miranda 方法。各位道友可以自行编写一个用 abstract 修饰的 Java 类，使其实现某个接口类，同时不实现接口方法，然后借助于 HSDB 来观察 Java 类在 JVM 内部所对应的 instanceKlassOop 对象实例的 _vtable_len 大小，从而验证 HotSpot 上面这段逻辑。

在 klassVtable.cpp::compute_vtable_size_and_num_mirandas() 函数中调用 klassVtable::get_num_mirandas() 函数，而后者调用 klassVtable::get_mirandas() 函数完成所有 miranda 方法的搜索和判断。在该函数中分别对当前 Java 类的全部接口类进行遍历，而在遍历每一个接口类时，又对该接口类所继承的父类接口类进行遍历，如此一层一层往上搜索全部 miranda 方法：

清单：/src/share/vm/oops/klassVtable.cpp

作用：get_mirandas() 方法

```

void klassVtable::get_mirandas(GrowableArray<methodOop>* mirandas,
                               klassOop super, objArrayOop class_methods,
                               objArrayOop local_interfaces) {
    assert((mirandas->length() == 0) , "current mirandas must be 0");

    // 遍历当前 Java 类所实现的全部接口类
    int num_local_ifs = local_interfaces->length();
    for (int i = 0; i < num_local_ifs; i++) {
        instanceKlass *ik = instanceKlass::cast(klassOop(local_interfaces->obj_at(i)));
        add_new_mirandas_to_list(mirandas, ik->methods(), class_methods, super);

        // 遍历每一个接口类所继承的父接口类
        objArrayOop super_ifs = ik->transitive_interfaces();
        int num_super_ifs = super_ifs->length();
        for (int j = 0; j < num_super_ifs; j++) {
            instanceKlass *sik =

```

```
instanceKlass::cast(klassOop(super_ifs->obj_at(j)));  
    add_new_mirandas_to_list(mirandas, sik->methods(), class_methods, super);  
}  
}  
}
```

miranda 方法其实并不是 JVM 的一个标准规范,但是笔者本人实在是被这个机制所感动,冰冷的机器里面也是充满情义的,因此忍不住将其介绍给大家。

对于接口方法,Java 类中还会保存一种内部结构——itable,顾名思义,就是接口方法表。itable 主要用于接口类方法的分发,其机制与 vtable 类似,都会在运行期进行方法的动态绑定。各位有兴趣的道友可以自行研究。

8.7.7 vtable 特点总结

前文对 vtable 进行了比较全面的研究和验证,这里再次总结下其特点:

- ◎ vtable 分配在 instanceKlassOop 对象实例的内存末尾。
- ◎ 所谓 vtable,可以看作是一个数组,数组中的每一项成员元素都是一个指针,指针指向 Java 方法在 JVM 内部所对应的 method 实例对象的内存首地址。
- ◎ vtable 是 Java 实现面向对象的多态性的机制,如果一个 Java 方法可以被继承和重写,则最终通过 invokevirtual 字节码指令完成 Java 方法的动态绑定和分发。事实上,很多面向对象的语言都基于 vtable 机制去实现多态性,例如 C++。
- ◎ Java 子类会继承父类的 vtable。
- ◎ Java 中所有类都继承自 java.lang.Object, java.lang.Object 中有 5 个虚方法(可被继承和重写):

void finalize()

boolean equals(Object)

String toString()

int hashCode()

Object clone()

因此,如果一个 Java 类中不声明任何方法,则其 vtable 的长度默认为 5。

- ◎ Java 类中不是每一个 Java 方法的内存地址都会保存到 vtable 表中。只有当 Java 子类中声明的 Java 方法是 public 或者 protected 的,且没有 final、static 修饰,并且 Java 子类中的方法并非对父类方法的重写时,JVM 才会在 vtable 表中为该方

法增加一个引用。

- ◎ 如果 Java 子类某个方法重写了父类方法,则子类的 vtable 中原本对父类方法的指针引用会被替换为对子类的方法引用。

8.7.8 vtable 机制逻辑验证

上文一直在讲 Java 中实现多态是通过 vtable 这个机制,但是 vtable 到底是如何实现多态机制的呢?这在 JVM 内部颇为复杂,这里先对其进行逻辑上的推理,让各位道友可以在不用知道具体源码实现细节的前提下,也能够知道如何“玩转”多态。

其实,多态实现的机制也很简单,就是通过 vtable (貌似这句话白说了哈哈)。文字描述起来够费劲,各位道友理解起来也费劲,不如举个例子:

清单: Animal.java

作用: vtable 机制验证

```
public class Animal {
    public void say(){
        System.out.println("I'm animal");
    }

    public static void main(String[] args){
        Animal animal = new Dog();
        run(animal);

        animal = new Animal();
        run(animal);
    }

    public static void run(Animal animal){
        animal.say();
    }
}

class Dog extends Animal{
    @Override
    public void say() {
        System.out.println("I'm a dog");
    }
}
```

本示例在本节开始讲解 vtable 的时候提到过,不过稍微做了一点修改。在本示例中,有父类 Animal 和子类 Dog,子类 Dog 重写了父类的 say()方法。

在 `main()` 主函数中, 将 `animal` 引用分别指向 2 个不同的实例, 并调用 `run(Animal)` 方法。运行这个程序, 结果也如预测的那样, 如下:

```
I'm a dog
I'm animal
```

这样的输出结果充分演绎了面向对象语言所具有的多态特性, 而多态得以实现的奥秘, 其实就隐藏在 `vtable` 中。

根据前文所讲的 `vtable` 的构成原理, 类 `Animal` 的 `vtable` 的长度应该为 6, 除了所继承的 `java.lang.Object` 中的 5 个虚方法 (即可被重写的方法) 外, 其自身仅包含 1 个虚方法。并且其 `vtable` 中的第 6 个指针元素指向 `say()` 方法在 JVM 内部所对应的 `method` 实例对象的内存地址。同理, 子类 `Dog` 的 `vtable` 的长度也应该等于 6, 因为前文讲过, 子类会完全继承父类的 `vtable`, 并且如果子类重写了父类的方法, 则 JVM 会将子类 `vtable` 中原本指向父类方法的指针成员修改成重新指向子类的方法。

类 `Animal` 和子类 `Dog` 的 `vtable` 结构分别如图 8.16 中左图和右图所示。

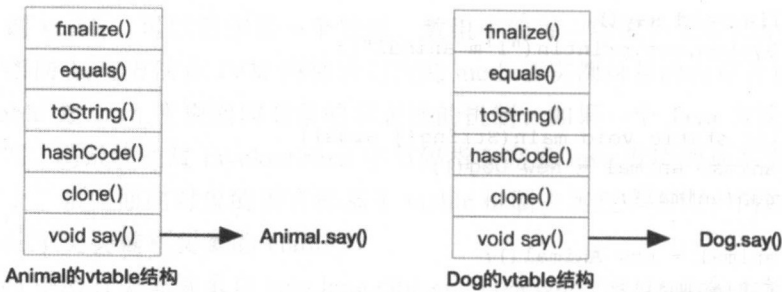


图 8.16 `Animal` 和 `Dog` 类的 `vtable` 结构

在 JVM 运行期, 会根据对象引用所执行的实际的实例调用实例的方法。不过, 这首先得从编译器说起。上面示例中 `Animal.run(Animal)` 方法所对应的字节码指令如下 (使用 `javap` 命令显示):

```
public static void run(Animal);
Code:
  Stack=1, Locals=1, Args_size=1
  0:   aload_0
  1:   invokevirtual    #10; //Method say:()V
  4:   return
```

`Animal.run(Animal)` 方法的字节码指令主要的逻辑包含两步, 第一步是 `aload_0`, 第二步是 `invokevirtual`。字节码指令 `aload_0` 表示从第 0 个 `slot` 位置加载 Java 引用对象。由于

`Animal.run(Animal)`方法是一个 `static` 静态方法，其入参并没有隐式的 `this` 指针，所以 `slot` 中的第一个局部变量就是 `Animal.run(Animal)` 的第一个入参 `animal` 引用对象。接着第二步执行 `invokevirtual` 指令，Java 多态的秘密就隐藏在该指令后面所跟的操作数 `operand` 中。`invokevirtual` 指令后面的操作数是常量池的索引值，该值为 10，`javap` 命令显示索引为 10 的常量池所代表的字符串是 `Method say():V`，这表示 `invokevirtual` 在运行期调用的方法是 `void say()`。在运行期，JVM 将首先确定被调用的方法所属的 Java 类实例对象，JVM 会读取被调用的方法的堆栈，并获取堆栈中的局部变量表的第 0 个 `slot` 位置的数据，该数据一定是指向被调用的方法所属的 Java 类实例，原因很简单，凡是所对应的字节码指令为 `invokevirtual` 的 Java 方法，其必定是 Java 类的成员方法，而非静态方法，而 Java 类的成员方法的第一个入参一定是隐式的 `this` 指针，该指针就指向 Java 类的对象实例。同时，Java 类的第一个入参一定位于局部变量表的第 0 个位置，因此 JVM 可以从被 `invokevirtual` 所调用的方法的局部变量表中读取到 `this` 指针，从而得知被调用的 Java 类实例到底是哪一个。

以本示例为例，当 `animal` 引用变量指向 `new Dog()` 对象实例时，则 `say()` 方法的局部变量表第一个入参便指向 `new Dog()` 这个对象实例。同理，当 `animal` 引用变量指向 `new Animal()` 对象实例时，则 `say()` 方法的局部变量表的第一个入参便指向 `new Animal()` 实例对象。

JVM 获取到被 `invokevirtual` 指令所调用的方法所属的实际的类对象时，接着便能够通过对象获取到其对应的 `vtable` 方法分发表。`vtable` 表中保存当前类的每一个方法的指针。JVM 会遍历 `vtable` 中的每一个指针成员，并根据指针读取到其对应的 `method` 对象，判断 `invokevirtual` 指令所调用的方法名称和签名与 `vtable` 表中指针所指向的方法的名称和签名是否一致，如果方法名称和签名完全一致，则算是找到了 `invokevirtual` 所实际调用的目标方法，于是 JVM 定位到目标方法的第一条字节码指令并开始执行。如此便完成了方法在运行期的动态分发和执行。

还是以本示例为例说明，当 `animal` 引用变量指向 `new Dog()` 对象实例时，JVM 会遍历 `Dog` 类所对应的 `vtable` 表，并搜索其中名称为 “say” 且签名为 “void ()” 的方法，很显然，`Dog` 类中存在该方法，于是 JVM 最终执行 `Dog` 类的 `say()` 方法。同理，当 `animal` 引用变量指向 `new Animal()` 对象实例时，JVM 最终执行的就是 `Animal` 类中的 `say()` 方法。

事实上，C++ 的虚方法分发的原理与此完全类似，所不同的是 C++ 的 `vtable` 由编译器在编译期间完成构建，而 Java 类的 `vtable` 则由 JVM 在运行期进行构建。

8.8 本章总结

本章主要描述了 Java 方法解析的技术实现。相比于前面章节所讲解的 Java 类字段的解析，

Java 类方法解析明显要复杂得多。这种复杂性体现在 Java 方法属性本身拥有众多信息，尤其是字节码指令部分。除了字节码指令，还有 LVT、miranda 方法等，存储格式比较复杂，并且概念理解起来也并不是一件轻松的事情。JVM 为 Java 方法在内存中所构建的对等体也明显更加复杂。

同时，Java 方法的解析还承担了一部分实现面向对象机制的责任，其核心技术便是 vtable。只有真正理解了 vtable 的实现机制，才能真正理解 Java 面向对象与多重继承的原理。

第 9 章

执行引擎

本章摘要

- ◎ JVM 的取指与译码机制
- ◎ 栈顶缓存原理
- ◎ 操作数栈与栈帧重叠技术
- ◎ JVM 指令集特点与实现

可以这么说，在 JVM 内部，最精华的部分便是执行引擎（当然，GC 也绝对是精华）。几乎每一款 JVM 都在执行引擎上面下足了功夫，从最原始低效的字节码解释器，到模板解释器，再到 JIT 即时编译器，许多大神提出了各种优化策略和理论来提升 Java 程序的执行速度。有些公司出品的 JVM 直接将 Java 程序翻译成本地机器码，而在安卓体系中，也使用了 AOT 技术来提升运行时效率。

JVM 的执行引擎本身是一个相当复杂深奥的模型，对于从来没有涉及过底层的广大程序员而言，想理解它十分困难。笔者力图使用比较浅显的语言来将 JVM 执行引擎实现的技术细节清楚地描述出来，使广大道友不仅仅局限于理论研究，而是能够真正一窥其具体的技术内幕，不仅能够闻其道，而且能够知其术，做到知行合一。往往越是高深精妙的理论，如果仅仅研究理论，往往越容易让人迷糊，而配合着源码看，便能真正理解这些深奥的理论。

本书在描述 Java 执行引擎时，将试图对照物理机器 CPU 的执行机制，例如指令集、取指机制、程序计数器等，从物理机器 CPU 执行的角度看 Java 的执行引擎，通过这样的对比，希望能够让各位道友不要陷入 JVM 那复杂深奥的引擎模型里面，而是能够站在一个较高的角度看待问题。

JVM 执行引擎毕竟只是个虚拟系统，本身并不具备真正的运算能力，其内部其实仍然需要依靠物理 CPU 才能完成运算功能，而物理 CPU 仅识别二进制机器指令，JVM 执行引擎既然需要依赖物理 CPU，就必然需要将字节码指令最终转换为二进制机器指令，因此下文在讲解 JVM 执行引擎的过程中，将不可避免地涉及汇编语言。而这也符合本书的宗旨——重点讲解 JVM 内部的具体技术实现，而非主要讲理论。

9.1 执行引擎概述

所谓执行引擎，就是一个运算器，能够识别所输入的指令，并根据输入的指令执行一套特定的逻辑，最终输出特定的结果。其实，相比于 JVM 这类虚拟机而言，物理实体机器也是有其特定的执行引擎的，物理机器的执行引擎便是 CPU（中央计算单元）。CPU 能够识别机器指令，并根据机器指令完成特定的运算。

物理 CPU 执行指令的流程是这样的：

（1）取指。CPU 的控制器从内存读取一条指令并放入指令寄存器。物理机器指令一般由操作码和操作数组成，当然并不是所有的操作码都会有操作数。例如 `mov ax, 1` 这条机器指令，其中 `mov ax` 就是操作码，而 `1` 就是操作数，在 Intel 处理器上，这条指令所对应的十六进制数是 `0xB8 01`。

（2）译码。指令寄存器中的指令经过译码，确定该指令应进行何种操作（由操作码决定），操作数在哪里（由操作数决定）。

（3）执行。分两个阶段，“取操作数”和“进行运算”。

（4）取下一条指令。修改指令计数器（亦称程序计数器），计算下一条指令的地址，并重新进入取指、译码和执行的循环。

只要操作系统一启动，CPU 便会一直循环往复地执行上述流程，当机器啥事也不干的时候，会进入“空转”的状态，但是并不会停止。这种机制说白了与汽车发动机一样，只要汽车一启动，发动机就会一直转下去，没挂档位的时候也会保持空转状态，如果发动机不转了汽车就熄火了。类似的机制还有很多，例如视窗系统会有一个静默线程一直保持无限的 `while` 循环，当收到外部消息（例如鼠标点击）时就对消息进行处理，如果一直没有收到消息就一直循环下去，以此来确保视窗程序一直运行下去。Web 服务器也是一样，会有一个线程一直保持循环，如果接收到客户端请求就启动新的线程/进程处理。

JVM 既然作为虚拟机，自然也得有这么一套虚拟的 CPU 执行机制，按照“取指→译码→执行→取下一条指令”这一流程循环往复地执行下去。不过 JVM 不像真正的操作系统那样，当

没事可干的时候让 CPU 保持空转，如果 JVM 所运行的 Java 程序执行完了，Java 程序的生命周期会终止，而 JVM 虚拟机本身也会退出。所以，如果想让 JVM 一直保持“空转”，只能在 Java 程序里的某个线程中一直保持空循环。Tomcat 这款 Web 应用服务器程序便是这种机制，否则一旦没有外部 http 请求过来，Tomcat 程序及其宿主 JVM 虚拟机都会“寿终正寝”。类似的，Hadoop、Spark 之类的分布式系统亦都有类似机制。

正因为 JVM 没有空转机制，因此 JVM 一旦启动，处理完自身的初始化逻辑，便会进入 Java 程序，执行 Java 的字节码指令。前文讲过，JVM 进入 Java 程序之前，会先确定 Java 程序的 main() 主函数及其所在的类，加载 Java 主类并执行 main() 主函数。在 JVM 调用 Java 的 main() 主函数的链路上，会经过 CallStub 例程和 zerolocals 例程，在 zerolocals 例程中，JVM 会为 Java main() 主函数创建栈帧，创建完栈帧，最终 JVM 会调用如下逻辑：

清单：/src/cpu/x86/vm/templateInterpreter_x86_32.cpp

功能：JVM 调用 Java 字节码指令

```
address InterpreterGenerator::generate_normal_entry(bool synchronized) {

    // ...
    address entry_point = __ pc();

    // ...

    // 创建栈帧
    generate_fixed_frame(false);

    // ...

    // 跳转到目标 Java 方法的第一条字节码指令，并执行其对应的机器指令
    __ dispatch_next(vtos);

    //...
    return entry_point;
}
```

InterpreterGenerator::generate_normal_entry(bool synchronized)函数在前文讲解 Java 方法的栈帧时，已经分析了其一部分逻辑，其中，详细地分析了这个函数中的 generate_fixed_frame()函数，Java 方法栈帧的创建便是通过该函数实现的。当 JVM 调用 Java 主函数 main()时，便是为 main()主函数创建栈帧，创建完栈帧，接着又有一系列逻辑处理（例如，方法校验、调用计数等），最后会执行这个函数中的__dispatch_next(vtos)函数（准确地说，函数前面的__是一个宏，为了简化，这些细节就不赘述了，有细节控的道友大可忽略）。从这个函数开始，JVM 将读取到 Java 主函数的第一条字节码指令，并执行第一条字节码指令所对应的机器指令，并由此进入

“轰轰烈烈”的 Java 程序的世界中去。__dispatch_next(vtos)函数是一个平台相关的函数，在 32 位 x86 平台上，其对应的实现如下：

清单：/src/cpu/x86/vm/interp_masm_x86_32.cpp

功能：dispatch_next()函数

```
void InterpreterMacroAssembler::dispatch_next(TosState state, int step) {
    load_unsigned_byte(rbx, Address(rsi, step));
    increment(rsi, step);
    dispatch_base(state, Interpreter::dispatch_table(state));
}
```

现在完全看不懂这个实现逻辑，下文会慢慢道来。其实，dispatch_next()函数是 JVM 内部非常核心的一个函数，该函数的主要功能就是进行“取指”。前面刚刚讲过，JVM 虚拟机与真实的物理机器执行指令的流程完全一样，都是循环往复地执行“取指→译码→执行→取指”的过程，下面便围绕这个过程，详细描述 JVM 内部取指、译码和执行的实现细节。

9.2 取指

在研究 JVM 的“取指”机制之前，先了解下物理机器级别的取指方式。对于直接运行在物理机器上的软件程序，其经过编译后直接形成二进制的物理机器指令编码。当操作系统加载这个软件程序时，会在内存中为该程序创建如图 9.1 所示的数据。

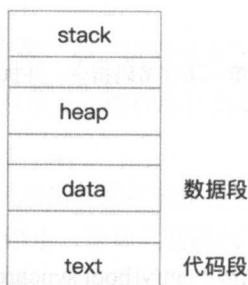


图 9.1 程序被操作系统加载后的内存映像

在一个基于段式内存管理的架构中，当操作系统将程序加载进内存之后，会将程序编译后的二进制代码指令存储到一个专门的区域——代码段（即图 9.1 中最下面的区域）。操作系统执行程序的过程，就是将代码段中的指令取出来逐个执行的过程。另外操作系统会将程序中的静态字段存储到数据段中，并为程序初始化堆栈空间，这部分内容超出了本书范围，有兴趣的道友可以另外自行研究操作系统编译原理。

操作系统会将软件程序的二进制机器码指令全部读进代码段中，当操作系统开始执行该软件程序时，CPU 便会读取代码段中的第一条机器指令，然后进入译码→执行→取下一条指令→译码→执行的循环，直到执行完该程序。

CPU 在取指时，先从程序的代码段中读取出操作码，在译码阶段，译码逻辑会判断该操作码，并从代码段中读取其所对应的操作数。例如，假设软件程序的二进制机器指令中有一条指令是 `mov ax, 3`，则 `mov ax` 是一个操作码，其操作数是 3。在取指时，CPU 首先读入 `mov ax` 这条操作码，译码器识别出该指令后面会跟着一个 32 位的数字（假设在 32 位 x86 平台上），于是译码器会接着从代码段内存区读取跟随在 `mov ax` 这条指令后面的操作数 3，如此完成整条指令的读取，接着交给运算器执行。当然，事实上 CPU 取指、译码和执行的机制是非常复杂的，这里将其简化，只是为了说明 CPU 的工作原理。

通过这个物理机器取指和译码的过程可知，物理机器要完成软件程序的运行，得具备 2 个条件：

- ◎ 内存中要存储软件程序编译后的指令。
- ◎ CPU 要能够识别出代码段中的指令和操作数。

对于第一点，操作系统运行软件之前，会将其源代码所对应的机器指令全部读进内存区。Java 程序启动后，JVM 也会将 Java 源代码所对应的字节码指令全部读进 JVM 内存中。而对于第二点，在我们的想象中，物理机器内部似乎要维护一张大而全的指令集表，每次 CPU 读入一个机器指令时，去扫描这张指令集表，从而识别出所读取到的机器码，并进一步判断该操作码后面是否有操作数。而事实上，物理机器内部这张所谓的指令集表并非仅仅是内存中的数据那么简单，物理机器“内置”的指令集其实是硬件结构，更具体地说是数字电路（呵呵，讨论得有点深，不过技多不压身啊），这些数字电路被集成进 CPU 内部，只要向 CPU 传递一个指令（0 和 1 的组合），CPU 就会依据其预先设定好的电路进行解码（高低电平），然后操作对应的寄存器或者某些电路去读取该指令操作码后面的操作数。同时，另一些电路则会被触发读取当前机器指令的下一条指令，如此一来，CPU 便能完成“取指→译码→执行→继续取指”的循环了。这便是 CPU 识别并执行机器指令的原理了。

在这一点上，JVM 基本也继承了这一思想（事实上想不继承都难，除非能够提出一种完全异于冯·诺依曼体系结构的数字计算机）。JVM 作为一款虚拟机，有其自己的一套指令集，这套指令集必定能够被 JVM 的虚拟运算器所识别，但是 JVM 并没有真正的硬件译码电路来识别 JVM 的这套指令，因此只能使用软件模拟。这种软件模拟的结果就是 JVM 需要使用软件的方式在内存中维护一套指令集，否则 JVM 无法识别 Java 方法所对应的指令操作码。这套指令集包含在下面的代码文件中：

清单：/src/share/vm/interpreter/bytecodes.hpp

功能：JVM 指令集定义

```
enum Code {
    _illegal                = -1,

    // Java bytecodes
    _nop                    = 0, // 0x00
    _aconst_null            = 1, // 0x01
    _iconst_m1              = 2, // 0x02
    _iconst_0               = 3, // 0x03
    _iconst_1               = 4, // 0x04
    _iconst_2               = 5, // 0x05
    _iconst_3               = 6, // 0x06
    _iconst_4               = 7, // 0x07
    _iconst_5               = 8, // 0x08
    _lconst_0               = 9, // 0x09
    _lconst_1               = 10, // 0x0a
    _fconst_0               = 11, // 0x0b
    _fconst_1               = 12, // 0x0c
    _fconst_2               = 13, // 0x0d
    _dconst_0               = 14, // 0x0e
    _dconst_1               = 15, // 0x0f
    _bipush                 = 16, // 0x10
    _sipush                 = 17, // 0x11
    _ldc                    = 18, // 0x12
    _ldc_w                  = 19, // 0x13
    _ldc2_w                 = 20, // 0x14
    _iload                  = 21, // 0x15
    _lload                  = 22, // 0x16
    _fload                  = 23, // 0x17
    _dload                  = 24, // 0x18
    _aload                  = 25, // 0x19
    _iload_0                = 26, // 0x1a
    //...
}
```

这个枚举中定义了 JVM 的全部指令集，比如你熟悉的 `iload`、`iconst_0` 之类的，都包含在内。由于这些指令操作码定义在 C++ 的枚举类中，因此在操作系统加载 JVM 时便会将这些指令集读进内存之中，这便是 Java 虚拟机的执行引擎赖以运行的基础。在 JVM 运行期，Java 字节码的译码系统完全依赖于这套“软指令集”。

9.2.1 指令长度

既然 JVM 内部定义了这么一套指令集，是否就能完成执行引擎的功能呢？很显然，答案是不能的。仅有这么一套指令集，JVM 无法据此执行字节码指令。别说执行字节码指令，便连取指的功能都完成不了，为何？前面说过，物理机器 CPU 在读取机器指令时，会先读取指令中的操作码，CPU 会识别出操作码并据此判断操作码后面是否跟随有操作数。CPU 只有知道一个操作码后面是否跟随操作数，以及所跟随的操作数的大小，才能计算出下一条指令的位置，从而完成“取指”的功能。举个例子，例如一个 C 程序编译后包含下面 3 条机器指令：

```
mov ax, 1
mov ax, 2
mov ax, 3
```

这 3 条指令所对应的十六进制机器码如下：

```
0xB8 01
0xB8 02
0xB8 03
```

当 CPU 执行到第一条指令时，先读取 0xB8 这个操作码（这个操作码表示 `mov ax`），CPU 的译码电路“翻译”出这个操作码，并知道其后面会跟着一个操作数，该操作数的宽度是 32 位，CPU 据此便知道第二条指令的操作码的位置，第二条指令操作码的位置相对于第一条指令操作码的位置再往前移动 32 位，这是因为程序的机器码在内存中是连续存储的，于是 CPU 便驱动其内部的相关电路去内存中读出 `mov ax` 后面所跟随的操作数。同时，当 CPU 执行完第一条指令后，便能接着取下一条指令中的操作码，完成继续取指。

需要注意的是，对于计算机而言，无论操作码还是操作数，在内存中都只是一串 0 和 1 的组合而已（准确地说是一串高低电平），因此如果直接将一个数字交给 CPU，CPU 是无法知道这个数字所代表的到底是操作码还是操作数，例如 `mov ax` 这个操作码所对应的十六进制编码是 0xB8（Intel CPU），但是可能某个操作码后面所跟随的操作数也是 0xB8，因此如果直接将 0xB8 交给 CPU，CPU 是无法区分的。所以，当 CPU 在执行一段程序时，一定是先读取程序中的第一条指令的操作码并进行译码，计算该操作码后面是否跟随操作数以及操作数的数据宽度，如此 CPU 才能计算出下一条指令的起始位置并读取下一条指令中的操作码，然后继续译码，继续计算下一条指令的起始位置……，如此循环往复。这便是“取指”的关键所在。

妙也！

所以 CPU 要能够正确完成取指，不仅仅需要识别出操作码本身，还得知操作码后面所跟随的操作数。

由于 JVM 完全继承了这一设计思想，因此也必须规定出每一条字节码指令后面所跟随的操

作数及操作数的大小。很显然，上面在 `bytecodes.hpp` 中所定义的一套指令集并无这种规范，所以 JVM 必然在别的地方定义了这种规范，如：

清单：/src/share/vm/interpreter/bytecodes.cpp

功能：JVM 指令集定义

```
void Bytecodes::initialize() {
    // ...
    // bytecode      bytecode name    format   wide f.   result tp  stk traps
    def(_nop          , "nop"          , "b"     , NULL    , T_VOID   , 0, false);
    def(_aconst_null  , "aconst_null" , "b"     , NULL    , T_OBJECT , 1, false);
    def(_iconst_m1    , "iconst_m1"   , "b"     , NULL    , T_INT    , 1, false);
    def(_iconst_0     , "iconst_0"    , "b"     , NULL    , T_INT    , 1, false);
    def(_iconst_1     , "iconst_1"    , "b"     , NULL    , T_INT    , 1, false);
    def(_iconst_2     , "iconst_2"    , "b"     , NULL    , T_INT    , 1, false);
    def(_iload_1      , "iload_1"     , "b"     , NULL    , T_INT    , 1, false);
    //....
}
```

在该初始化函数中，JVM 定义了每个 `bytecode` 的名字（字符串）、`format`、字节码的返回结果 `result` 类型等。可是纵观这张表，并没有哪里明确指出每个字节码指令后面是否跟随操作数及操作数的宽度。其实秘密就藏在 `format` 这一列中，这一列记录了每一个字节码指令的总长度。例如，`_iconst_0` 这个字节码的 `format` 是 `b`，则表示该字节码指令，包括操作码和操作数，其总长度一共只有 1，由此可以推测，该字节码指令其实是没有操作数的。这里所谓的“长度为 1”，是指 1 字节，因为 Java 的每个字节码指令都仅占 1 字节，这也是为何 Java 字节码指令数量少于 256 个的原因。

再如 `bipush` 这条指令对应的 `format`=“bc”，则表示 `bipush` 这个操作码后面会跟一个宽度为 1 字节的操作数。同理，`sipush` 对应的 `format`=“bcc”，则表示 `sipush` 指令后面会跟一个宽度为 2 字节的操作数。这很好理解，因为 `bipush` 指令表示将一个 1 字节的数据推送至操作数栈栈顶，因此该指令后面的操作数仅占 1 字节，而 `sipush` 表示将一个 `short` 类型的操作数推送至栈顶，而一个 `short` 类型的数据占 2 字节。

既然推送一个 `short` 类型的数据至栈顶的字节码指令是 `sipush`，那么推送一个 `int` 类型的数据至栈顶的字节码指令是什么呢？直接看上面的字节码指令表是看不出来的，不过可以通过试验来验证：

清单：Test.java

功能：测试将 `int` 类型数据推送至栈顶的字节码指令

```
public static void tdd(){
    int a = 3;
}
```


该程序定义了一个 `int` 类型的变量 `a`，并赋初值为 3。由于为变量赋值的过程必定会有压栈和出栈的操作，并且为变量所赋的值直接是一个自然数，因此编译后的代码中一定会包含将整型变量推送至栈顶的字节码指令。

编译该程序，并使用 `javap` 命令分析编译后的字节码文件，结果输出如下：

```
public static void tdd();
Code:
    Stack=1, Locals=1, Args_size=0
    0:   iconst_3
    1:   istore_0
    2:   return
```

从分析结果可以看出，编译器使用 `iconst_3` 这条字节码指令将自然数 3 推送至栈顶。在刚才 `Bytecodes::initialize()` 函数中所定义的字节码指令格式表中找到 `iconst_3`，可以看到其 `format="b"`。这说明 `iconst_3` 指令的总长度为 1，同时说明该操作码后面并没有操作数跟随。

在这里可以看到，自然数 3 虽然在源代码中被定义为整型，但是编译器直接使用一个特殊的字节码指令将其推送至栈顶，并没有使用“操作码 + 操作数”的方式来推送。JVM 如此设计的原因在于减小 Java class 的体积，字节码文件的体积减小了，其加载到内存后所占的内存空间也会降低，而 JVM 所牺牲的仅仅是在内存中多定义了一个字节码指令，以及为此多写了一段译码逻辑而已。这种牺牲是值得的，并且是非常划算的，毕竟字节码指令只定义了一次，但是 JVM 会加载成千上万个 Java 字节码文件，如果每一个字节码文件中都包含一个 `int a = 3` 这样的逻辑，并且假设 JVM 使用“操作码+操作数”的方式来进行译码，那么每一条 `int a = 3` 这样的逻辑所对应的字节码，相比于 `iconst_3` 这套指令而言，除了字节码指令本身，还会额外多出来一个操作数，而一个操作数至少占 1 字节，虽然这一点内存空间不算啥，但是当 JVM 加载了成千上万个 Java 类时，这些多出来的内存空间累加起来就非常可观了。

事实上，JVM 为了节省空间，专门定义了 `iconst_0 ~ iconst_5` 这 6 条字节码指令，当将自然数 0~5 推送至栈顶时，便会分别生成这 6 条指令。其实这也是一种权衡，事实上 JVM 可以为每一个仅占 1 字节的数字分别定义一个特殊的字节码指令和译码逻辑，但是这样一来，指令和译码逻辑反而显得太臃肿，反而不美。

如果推送的整数大于 5，JVM 会如何处理呢？很简单，修改上面的 Test 类的 `tdd()` 方法中的 `a` 变量初始值，将其改成 6 看看。修改后编译 Java 类，并使用 `javap` 命令分析字节码文件，分析结果如下：

```
public static void tdd();
Code:
    Stack=1, Locals=1, Args_size=0
    0:   bipush 6
```

```

2:   istore_0
3:   return

```

可以看到，现在推送至栈顶的指令变成 `bipush 6` 了。现在的字节码指令格式终于变成了“操作码+操作数”这种范式。前面已经分析过，`bipush` 字节码指令的 `format=“bc”`，其长度为 2，表示 `bipush` 操作码后面会跟随一个只占 1 字节码宽度的操作数。那么如果要将一个宽度超过 1 字节的整数推送至栈顶呢？很简单，继续试验，将上面测试用例 `Test.tdd()` 方法中的变量 `a` 的初始值改成 300，编译后的字节码指令如下：

```

public static void tdd();
Code:
    Stack=1, Locals=1, Args_size=0
    0:   sipush 300
    3:   istore_0
    4:   return

```

可以看到，现在推送至栈顶的指令变成了 `sipush`，该指令前面也讲过，其 `format=“bcc”`，表示该指令后面会跟随 1 个占 2 字节宽度的操作数。

事实上，如果将上面的 `int a=3` 改成 `char a=3` 或者 `short a=3`，最终所生成的字节码指令也是 `iconst_3`。同样地，若将 `int a=300` 改成 `short a=300`，所生成的字节码指令与 `int a=300` 所生成的一样，都是 `sipush 300`。

由此可以知道，其实 JVM 体系对内存空间的使用标准非常严格，从编译期便开始进行优化，能使用 1 个操作码完成的事，就绝不使用由 1 个操作码+1 个占 1 字节宽度的操作数所组成的指令去完成；能够使用由 1 个操作码+1 个占 1 字节宽度的操作数所组成的指令去完成的事，也绝不使用由 1 个操作码+1 个占 2 字节宽度的操作数所组成的指令去完成。

大善！

接着往下分析，如果要推送的整数比较大，超过了 2 字节的宽度，编译器会生成什么样的指令呢？这也是上面未验证完的问题。2 字节所能表示的最大无符号整数是 65535，那么如果在 `Test.tdd()` 方法中这样定义变量 `a`：

```
int a = 65539;
```

所生成的字节码指令会是什么呢？

编译 `Test` 类并使用 `javap` 命令分析，输出如下：

```

public static void tdd();
Code:
    Stack=1, Locals=1, Args_size=0
    0:   ldc #6; //int 65539
    2:   istore_0

```

```
3:    return
```

这一次的指令终于有了很大的变化，变成了 `ldc #6`。所谓 `ldc`，全称是 `load constant`，意思是从常量池中加载（讲真，JVM 内部大部分名称都起得很直观明了，让人能够顾名思义，即使是缩写也是如此，但 `ldc` 是个例外，乍一看，真猜不出啥意思）。而其后面所跟随的操作数 `#6`，其实正是 `65539` 这个数字在字节码文件的常量池中的索引号。由此可见，当一个 `int` 型整数的宽度超过 2 字节时，Java 编译器便会将其直接编译进字节码文件的常量池中，而常量池中的数据在被 JVM 加载之后，会保存进 JVM 的常量区内。如果一个整数被保存进 JVM 的常量区之中，当其他 Java class 字节码文件中也使用了同样的整数为变量赋值时，则 JVM 不会重复将该整数写入常量区。JVM 通过这种方式避免大数据（虽然只占 4 字节）的内存重复占用。

事实上，只要为整型变量赋值的自然数超过 32767，Java 编译器便会使用 `ldc` 字节码指令编译源代码，而非 `sipush`，这是因为如果存在负数，第一位将会用于表示符号。同理，只要为整型变量赋值的自然数超过 127，Java 编译器也会生成 `sipush` 字节码，而非 `bipush`，这同样是因为第一位需要用于表示符号。看下面的示例：

清单：Test.java

功能：测试 `iconst`、`bipush`、`sipush` 和 `ldc`

```
public static void tdd(){
    int a = 5;
    int a2 = 6;

    int b = 127;
    int b2 = 128;

    int c = 32767;
    int c2 = 32768;
}
```

编译后使用 `javap` 命令分析，结果如下：

```
public static void tdd();
  Code:
    Stack=1, Locals=6, Args_size=0
    0:    iconst_5
    1:    istore_0
    2:    bipush   6
    4:    istore_1
    5:    bipush  127
    7:    istore_2
    8:    sipush  128
   11:    istore_3
   12:    sipush  32767
```

```

15:  istore  4
17:  ldc #2; //int 32768
19:  istore  5
21:  return

```

注意看 `int a=5` 和 `int a2=6` 所生成的字节码指令分别是 `iconst_5` 和 `bipush 6`，这是因为 JVM 规范只提供了 `iconst_0~iconst_5` 这 6 个特殊的字节码指令，当超过 6 时，并没有提供类似于 `iconst_6` 这样的字节码指令。而 `int b=127` 和 `int b2=128` 所生成的字节码也不同，分别是 `bipush 127` 和 `sipush 128`，`int c=32767` 和 `int c2=32768` 所生成的字节码也不同，分别是 `sipush 32767` 和 `ldc #2`，这是因为有一个二进制位需要用作区分正负数，因此 8 个二进制位最大只能表示到 127，16 个二进制数最大只能表示到 32767。

继续回到 `ldc` 这个字节码指令。既然 `ldc` 指令后面所跟随的内容是常量池的索引，而非真正的操作数，那么来看看该指令在 JVM 内部的格式定义，如下：

```
def(_ldc, "ldc", "bk", NULL, T_ILLEGAL, 1, true );
```

其 `format="bk"`，长度是 2 位，这意味着 `ldc` 指令后面只能跟随一个宽度为 1 字节的操作数。

不过聪明的你可能会马上想到，既然 `ldc` 指令后面所跟随的操作数是常量池的索引，并且这个操作数只能占 1 字节的宽度，但是 1 字节所能代表的最大数是 256，那么如果常量池很大，其中的元素超过 256 个，那么 `ldc` 这个指令岂不是会存在问题了吗？为了分析问题，下面再次改造 `add()` 方法，改造后的程序如下：

清单：/Test.java

功能：测试 `ldc` 指令

```

public static void tdd(){
    int v1=32769; int v2=32770; int v3=32771; int v4=32772; int v5=32773;
    int v6=32774; int v7=32775; int v8=32776; int v9=32777; int v10=32778;
    int v11=32779; int v12=32780; int v13=32781; int v14=32782; int v15=32783;
    int v16=32784; int v17=32785; int v18=32786; int v19=32787; int v20=32788;

    //这里再定义若干 int 型变量，其初始值都超过 32767，并且各个变量的初始值彼此不同
    //...

    int eeef11=46779; int eeef12=46780; int eeef13=46781; int eeef14=46782;
    int eeef15=46783;
    int eeef16=46784; int eeef17=46785; int eeef18=46786; int eeef19=46787;
    int eeef20=46788;

    //一共定义超过 256 个局部整型变量
}

```

由于 `Test.tdd()` 方法内部定义了超过 256 个整型变量，值都大于 32767，并且彼此不同，因

此每一个给变量赋值的自然数都会在 Test 类所生成的字节码文件的常量池中的元素数组中占有一席之地，并且部分常量池元素的索引号会大于 256，这超过 1 字节所能表示的最大范围。且看这种情况下 ldc 字节码指令如何处理索引号大于 256 的元素加载。编译 Test 类，并使用 javap 命令分析编译后的字节码文件，输出如下：

```
$ javap -v Test
Compiled from "Test.java"
class Test extends java.lang.Object
  SourceFile: "Test.java"
  minor version: 0
  major version: 50
  Constant pool:
const #1 = Method      #302.#310;  // java/lang/Object."<init>":()V
const #2 = int      32769;
const #3 = int      32770;
const #4 = int      32771;
const #5 = int      32772;
//...此处省略若干常量池元素
const #253 = int 44781;
const #254 = int 44782;
const #255 = int 44783;
const #256 = int 44784;
const #257 = int 44785;
const #258 = int 44786;
const #259 = int 44787;
//...省略若干常量池内容

{
test1();
  Code:
    Stack=1, Locals=1, Args_size=1
    0:   aload_0
    1:   invokespecial    #1; //Method java/lang/Object."<init>":()V
    4:   return
  LineNumberTable:
    line 18: 0

public static void tdd();
  Code:
    Stack=1, Locals=300, Args_size=0
    0:   ldc #2; //int 32769
    2:   istore_0
    3:   ldc #3; //int 32770
    5:   istore_1
    6:   ldc #4; //int 32771
```

```

8:   istore_2
9:   ldc #5; //int 32772
11:  istore_3

//...省略若干字节码指令

1004: ldc #253; //int 44781
1006: istore 252
1008: ldc #254; //int 44782
1010: istore 253
1012: ldc #255; //int 44783
1014: istore 254
1016: ldc_w #256; //int 44784
1019: istore 255
1021: ldc_w #257; //int 44785
1024: istore_w 256
1028: ldc_w #258; //int 44786
1031: istore_w 257

```

//...省略若干字节码指令

从 `javap` 命令的输出结果可以看到, 常量池中的元素索引号已经超过 256。观察 `tdd()` 方法的字节码指令, 可以看到当使用自然数 44783 为 `tdd()` 方法内的局部变量赋值时所使用的字节码指令是 `ldc #255`, 而当使用自然数 44784 为 `tdd()` 方法内的局部变量赋值时所使用的字节码指令就变成了 `ldc_w`, 并且此后的局部变量的赋值指令都变成了 `ldc_w`。显然, 当从常量池中索引号大于 255 的地方取值时, JVM 使用了 `ldc_w` 指令。该指令的含义是: 将 `int`、`float` 或 `String` 型常量值从常量池中推送至栈顶 (宽索引)。看看 `bytecodes.cpp` 中如何定义该指令格式:

```
def(_ldc_w, "ldc_w", "bkk", NULL, T_ILLEGAL, 1, true );
```

其 `format="bkk"`, 总长度变成了 3, 说明 `ldc_w` 指令后面可以跟随一个 2 字节宽的操作数。如此一来就通了, 对于将常量池中索引号大于 255 的常量池推送至栈顶, JVM 就使用 `ldc_w` 指令, 反之就使用 `ldc` 指令。道理其实也很简单, 这是在尽量压缩字节码文件的体积。使用十六进制编辑器打开上面更改过的 `Test` 类的字节码文件, 可以查找到 `ldc #255` 和 `ldc_w #256` 这 2 条字节码指令的内容, 如图 9.2 所示。

```

169  12f8 36f7 12f9 36f8 12fa 36f9 12fb 36fa
170  12fc 36fb 12fd 36fc 12fe 36fd 12ff 36fe
171  1301 0036 ff13 0101 c436 0100 1301 02c4
172  3601 0113 0103 c436 0102 1301 04c4 3601
173  0313 0105 c436 0104 1301 06c4 3601 0513
174  0107 c436 0106 1301 08c4 3601 0713 0109
175  c436 0108 1301 0ac4 3601 0913 010b c436

```

图 9.2 观察 `ldc #255` 和 `ldc_w #256` 指令在字节码文件中的内容

由图 9.2 可以看到, ldc #255 指令在字节码文件中的内容是 0x12ff, 这是因为 ldc 指令的十六进制编码是 0x12, 而 255 所对应的十六进制数是 0xff。同理, ldc_w 指令所对应的十六进制编码是 0x13, 256 所对应的十六进制数是 0x0100, 所以 ldc_w #256 指令的十六进制内容是 0x130100。所以在字节码文件中, ldc 整条指令, 操作码连同操作数, 一共只占 2 字节; 而 ldc_w, 与 bytecodes.cpp 中所定义的 format 一致, 一共只占 3 字节。由此可见, JVM 为了尽可能地减小字节码文件的体积, 真所谓无所不用其极, 玄之又玄, 众妙之门! 而这种努力是非常值得的, 随便一个 Java Web 程序的 war 包中都包含成千上万个 Java class 字节码文件, 字节码文件的体积得以减少, 则在网络传输 (例如, 远程部署) 时将提升速度, 并且 JVM 将它们读进内存后所占内存空间也会减少。虽然一个字节码文件的体积的减少量并不是很明显, 但是宏观上的效应就很可观了。

纵观 bytecodes.cpp::initialize() 函数中所定义的全部字节码指令的格式, 会看到绝大多数字节码指令的 format 所包含的字符数都是 1 位, 少部分为 2 位和 3 位, 而超过 3 位的则更少, 寥寥无几。因此 JVM 的字节码指令集在设计上非常紧凑和简洁。有兴趣的道友可以继续研究对于 long 型和 double 型的变量赋值所对应的字节码指令, 并分析指令在字节码文件中所占用空间的大小。

上面讲了这么多, 除了可以知道对于 int 型变量的自然数赋值指令包含 iconst、bipush、sipush、ldc 和 ldc_w 这五种外, 最主要的是明白了 JVM 内部对字节码的指令宽度是有严格定义的, 而字节码指令的宽度对于 JVM 虚拟机完成取指是至关重要的, JVM 只有知道了每一个字节码指令所占的宽度, 才能完成“取指→译码→执行→取指”这种循环, 将程序一直执行下去。

Bytecodes::initialize() 函数会在 JVM 启动期间被调用, 该函数执行完成之后, 各个字节码指令所占的内存宽度会被 JVM 所记录, JVM 在运行期执行 Java 程序时会不断地读取该函数所维护的表, 计算每个字节码指令的长度。

9.2.2 JVM 的两级取指机制

前面分析了执行引擎取指的关键一步——计算每一个指令的总长度。无论是物理 CPU, 还是 JVM 的软件模拟的执行引擎, 其内在的核心机制都是类似的。在 HotSpot 内部也存在与 CPU 内部类似的译码器, HotSpot 里面通常叫作“解释器”。HotSpot 提供了好几种解释器, 例如字节码解释器 bytecodeInterpreter、模板解释器 templateInterpreter 等。如果 HotSpot 以模板解释器来执行字节码指令 (事实上这也是默认的方式), 则所有的字节码指令都会通过 TemplateInterpreterGenerator::generate_and_dispatch() 这个函数来生成对应的机器指令。在该函数中实现了指令跳转 (即取下一条字节码指令) 的逻辑。该函数的实现如下:

清单：/src/share/vm/interpreter/templateInterpreter.cpp

功能：generate_and_dispatch()函数中的取指逻辑

```
void TemplateInterpreterGenerator::generate_and_dispatch(Template* t,
TosState tos_out) {
    int step;
    if (!t->does_dispatch()) {
        step = t->is_wide() ? Bytecodes::wide_length_for(t->bytecode()) :
Bytecodes::length_for(t->bytecode());
        if (tos_out == ilgl) tos_out = t->tos_out();

        __ dispatch_prolog(tos_out, step); //该函数暂时无实现
    }

    // generate template
    t->generate(_masm);

    // advance--取下一条指令
    if (t->does_dispatch()) {
#ifdef ASSERT
        // make sure execution doesn't go beyond this point if code is broken
        __ should_not_reach_here();
#endif // ASSERT
    } else {
        // dispatch to next bytecode
        __ dispatch_epilog(tos_out, step); //取下一条字节码指令
    }
}
```

该函数会在 JVM 启动期间被调用，用于生成固定的取指逻辑。需要注意的是，JVM 会为每一个字节码指令都生成一个特定的取指逻辑，这是因为不同的字节码其指令宽度不同，因此取指逻辑也肯定不统一。该函数其实主要干了两件事：

(1) 为 Java 字节码指令生成对应的汇编指令。

(2) 实现字节码指令跳转，即“取指”。

这两件事对于 templateInterpreter 模板解释器而言，是核心中的核心。通过该函数也可以知道，HotSpot 内部在生成“取指”逻辑的同时，也会为字节码指令生成对应的本地机器码。或者换言之，HotSpot 在为每一个字节码指令生成其机器逻辑指令时，会同时为该字节码指令生成其取指逻辑（取其下一条指令）。该函数的第一个入参是 Template* 类型的指针，Template 便是解释器为每个 Java 字节码指令所定义的汇编模板，在本函数中通过调用 t->generate(_masm)来生成字节码指令的机器码，该逻辑会在下文中详细讲解，这里重点关注 TemplateInterpreterGenerator::generate_and_dispatch()函数中所调用的__dispatch_epilog(tos_out, step)这行代码。这行

代码便是在生成跳转（取指）逻辑。该函数的第二个入参是 `step`，`step` 是 Java 字节码指令的“步长”或所占的数据宽度，其单位是“字节”或 8 位。在 `TemplateInterpreterGenerator::generate_and_dispatch()` 函数中，通过 `step = t->is_wide() ? Bytecodes::wide_length_for(t->bytecode()): Bytecodes::length_for(t->bytecode())` 来计算出字节码指令的步长，而计算的逻辑正与上文所描述的一致，即根据 `Bytecodes::initialize()` 函数中为每个字节码指令所定义的 `format` 字段来计算，`format` 包含几个字符，则字节码的步长便是几。有兴趣的道友可以跟进 `Bytecodes::length_for()` 函数去看下具体的实现逻辑。

在 32 位 x86 平台上，`__dispatch_epilog(tos_out, step)` 函数的实现逻辑如下（由于其内部涉及汇编，因此一定是 CPU 平台相关的）：

清单：/src/cpu/x86/vm/interp_masm_x86_32.cpp

功能：取指逻辑演示

```
void InterpreterMacroAssembler::dispatch_epilog(TosState state, int step) {
    dispatch_next(state, step);
}

void InterpreterMacroAssembler::dispatch_next(TosState state, int step) {
    // 加载下一个字节码指令
    load_unsigned_byte(rbx, Address(rsi, step));
    // advance rsi
    increment(rsi, step);
    dispatch_base(state, Interpreter::dispatch_table(state));
}
```

在 `InterpreterMacroAssembler::dispatch_epilog()` 函数中调用 `dispatch_next()` 函数，而后者则通过“三部曲”完成两级取指逻辑。所谓两级取指逻辑，第一级是获取字节码指令，当前字节码指令执行完成后，JVM 必须能够自动获取到其下一条字节码指令，这样才能循环往复地执行下去。而第二级取指逻辑则是取字节码指令所对应的本地机器指令，当前字节码指令对应的机器码执行完成之后，JVM 必须要能够跳转到下一条字节码指令所对应的机器码。

`InterpreterMacroAssembler::InterpreterMacroAssembler()` 函数中的这三部曲分别是：

```
load_unsigned_byte(rbx, Address(rsi, step));
increment(rsi, step);
dispatch_base(state, Interpreter::dispatch_table(state));
```

这三条指令在不同的 CPU 平台上会生成不同的取指机器指令，在 32 位 x86 平台上生成如下 3 条机器指令：

```
movzbl 0x1(%esi),%ebx
inc %esi
jmp *_dispatch_table(,%ebx,state)
```

这 3 条机器指令中的第一条和第三条用于对本地机器码取指和跳转，其逻辑先不必理会，第二条 `inc %esi` 则用于取字节码指令，该指令由 `InterpreterMacroAssembler::InterpreterMacroAssembler()` 函数中的 `increment(rsi, step)` 代码生成，`increment(rsi, step)` 函数便是 Jvm 模板解释器用于取指的核心逻辑，该函数的功能是计算下一个即将执行的 Java 字节码指令的内存位置，而计算公式很简单：

下一个字节码指令的内存位置 = 当前字节码的位置 + 当前字节码指令所占的内存大小 (以字节计)

这种公式很好理解，对于一个 Java 方法，当 JVM 将其加载进内存后，会将该 Java 方法所对应的全部字节码指令存放于一块内存区域中，这些字节码指令彼此相邻，在内存里线性按序存储，所以相邻的 2 条字节码指令所对应的内存地址的偏移量便是前面一条字节码指令所占的内存大小。事实上物理机器指令的存储方式也是这样的，因此物理 CPU 在取指时也遵循同样的逻辑。该函数就像汽车中的发动机曲轴，通过它，JVM 才能沿着人们所编写好的 Java 逻辑一直往下运行下去。而在物理机器层面，CPU 也具有取指功能，只不过 CPU 的取指功能是实实在在的硬件电路实现，而 JVM 仅仅是软件模拟。

`increment(rsi, step)` 函数的第 1 个人参是 `rsi` 寄存器，该寄存器“总是”指向当前字节码指令所在的内存位置，第 2 个人参是 `step`，`step` 是 Java 字节码指令的步长（即字节码指令所占的内存大小）。`increment(rsi, step)` 函数最终生成的机器指令类似于 `rsi = rsi + step`，表示将当前字节码指令所在的内存位置加上字节码指令的步长，从而得到当前字节码指令的下一条字节码指令的内存位置，这便完成了 JVM 取指的逻辑。`step` 的计算方式前文讲过，在模板表中通过 `format` 这个字段指定每个字节码指令的步长，并在 `TemplateInterpreterGenerator::generate_and_dispatch()` 函数中实现步长计算的逻辑，逻辑很简单，就是获取 `format` 字符串的字符数。例如对于 `iload_0` 指令，该字节码指令的 `format = "b"`，其步长便是 1 字节。

由于不同的 Java 字节码指令的步长是不同的，因此最终所生成的本地机器码也有所不同，如果字节码指令的步长是 1（只有操作码而没有操作数），则生成的本地机器指令便是 `inc %esi`，该指令表示对 `esi` 增加 1 字节（因为一个步长为 1 的 Java 字节码指令所占的内存空间为 1 字节）；而如果字节码指令的步长超过 1，则生成的本地机器指令便是 `add $operand, %esi`，该指令表示对 `esi` 进行累加；假设字节码的步长为 2，则对应的指令是 `add $0x2, %esi`，表示对 `esi` 寄存器增加 2 字节。例如，对于 `iload_1` 这样的字节码指令，该指令没有操作数，因此步长为 1，那么 `iload_1` 字节码的下一个字节码的内存位置相对于 `iload_1` 偏移量是 1 字节，所以生成的机器指令是 `inc %esi`，其将 `esi` 寄存器（该寄存器指向当前字节码指令的内存位置）的值加 1，便得到 `iload_1` 这条指令的下一条指令所在的内存位置。

可能聪明的你会想到这样一个问题：`rsi` 寄存器总是指向当前字节码指令所在的内存位置，

这一句恐怕不对吧，例如，当 JVM 在运行 Java 程序 `main()` 主函数所对应的第一条字节码指令时，这个时候根本就不存在“上一条”字节码指令，那么 `rsi` 寄存器指向哪里呢？

事实上，在 JVM 调用 Java 的 `main()` 主函数时，会先调用 `generate_fixed_frame(false)` 函数来创建栈帧，而在这个过程中，JVM 便会有一个逻辑获取 Java 的 `main()` 主函数在 JVM 内部的第一条字节码指令，并将 `esi` 寄存器指向第一条字节码的位置。这个逻辑的具体实现在前面讲解 Java 函数栈帧的章节里详细讲解过，此处不再赘述。同时，关于 Java 方法在 JVM 内部的映射以及 Java 方法的字节码指令在 JVM 内部的存储，也在前面讲解 Java 方法的章节里详细讲解过，还不清楚的小伙伴可以去相关章节进行研究。

正因为 JVM 调用 Java 的 `main()` 主函数之前会先执行 `generate_fixed_frame(false)` 来创建栈帧，并在这个过程中将 `esi` 寄存器的值指向 `main()` 函数的第一条字节码指令，所以接着调用 `dispatch_next()` 函数时才能根据 `rsi` 进行偏移，顺利完成取指。

到了这里，终于可以与本章开始处所讲的 `InterpreterGenerator::generate_normal_entry()` 函数接上头了。本章一开始便讲到，在 JVM 进入 Java 世界之前，会先找到 Java 的 `main()` 主函数并调用，而调用 Java 主函数时，最终流程会进入 `InterpreterGenerator::generate_normal_entry()` 这个函数的逻辑中去。而在 `InterpreterGenerator::generate_normal_entry()` 函数中，除了会调用 `generate_fixed_frame()` 函数为 Java 的 `main()` 主函数创建栈帧之外（注意，在创建栈帧的过程中，JVM 会将 `rsi` 寄存器指向 `main()` 主函数的第一条字节码指令的内存地址），还将调用 `dispatch_next()` 函数执行 Java 的 `main()` 主函数的第一条字节码指令。在 32 位 x86 平台上，在 `InterpreterGenerator::generate_normal_entry()` 函数中所调用的 `dispatch_next()` 函数便是定义在 `/src/cpu/x86/vm/interp_masm_x86_32.cpp` 中的 `dispatch_next()` 这个函数。

当 JVM 第一次调用 Java 的 `main()` 主函数时，`rsi` 指向 Java 的 `main()` 主函数的第一条字节码指令在内存中的位置，但是从 `InterpreterGenerator::generate_normal_entry()` 函数中调用 `dispatch_next()` 函数时，仅传入了第一个参数 `tosState`，而第二个参数 `step` 并没有传递，因此 `step` 默认为 0，而当步长为 0 时，JVM 最终不会生成类似 `inc %esi` 这样的机器指令，即此时不会“取下一条指令”，因为 JVM 总得先执行第一条指令。只有第一条字节码指令执行完成之后，JVM 才能通过 `esi` 来获取下一条字节码指令所在的内存位置，从而挨个执行 Java 方法的全部字节码指令。

刚才讲过，JVM 内部其实存在两级取指逻辑，第一级取字节码指令，第二级取字节码指令对应的本地机器码。第二级的取指逻辑是通过 `InterpreterMacroAssembler::InterpreterMacroAssembler()` 函数中的第一和第三条指令完成的，这两条指令最终所生成的本地机器指令如下（32 位 x86 平台）：

```
movzbl step(%esi),%ebx
jmp    *_dispatch_table(,%ebx,state)
```

第一条指令 `movzbl step(%esi),%ebx` 取下一条字节码指令，并将其存储到 `ebx` 寄存器中，接着通过第二条指令跳转到下一条字节码指令所对应的本地机器码。注意，第二条指令中的 `_dispatch_table` 便是 JVM 内部所维护的跳转表，跳转表中记录了每个 JVM 字节码所对应的本地机器码实现，JVM 通过跳转表，完成第二级取指逻辑。在 JVM 取到某条字节码指令时，跳转到其对应的本地机器码并执行。

9.2.3 取指指令放在哪

前面讲过，如果 HotSpot 以模板解释器来执行字节码指令（事实上这也是默认的方式），则所有的字节码指令都会通过 `TemplateInterpreterGenerator::generate_and_dispatch()` 这个函数生成对应的机器指令，`TemplateInterpreterGenerator::generate_and_dispatch()` 函数主要完成两件事：

- （1）为当前字节码指令生成其对应的本地机器码。
- （2）为当前字节码指令生成其对应的取指逻辑（取下一条指令）。

这两件事分别通过 `t->generate(_masm)` 和 `__dispatch_epilog(tos_out, step)` 来完成，`TemplateInterpreterGenerator::generate_and_dispatch()` 函数先调用 `t->generate(_masm)` 为当前字节码指令生成其对应的本地机器码，具体生成逻辑在后文中讲解，而 `__dispatch_epilog(tos_out, step)` 的逻辑刚刚讲过，通过三部曲进行两级取指。`t->generate(_masm)` 和 `__dispatch_epilog(tos_out, step)` 这两个函数会分别向 JVM 内部的代码缓冲区中写入对应的本地机器指令，由此可知，字节码的取指逻辑其实是被写入到每一个字节码指令所对应的本地机器码所在内存的后面区域。其原因很简单，因为不同的字节码指令的步长不同，因此所生成的对 `rsi` 寄存器进行累加的逻辑也不同，所以不同字节码指令的取指逻辑肯定不同。事实上，这还与栈顶缓存有关。

使用 HSDIS 工具可以查看 JVM 模板解释器在运行期所生成的全部字节码指令的本地机器码。例如 `bipush` 的本地机器指令如图 9.3 所示。

```
bipush 16 bipush [0xb36d80a0, 0xb36d80e0] 64 bytes
```

```
[Disassembling for mach='i386']
0xb36d80a0: sub    $0x4,%esp
0xb36d80a3: fstps  (%esp)
0xb36d80a6: jmp    0xb36d80c4
0xb36d80ab: sub    $0x8,%esp
0xb36d80ae: fstpl  (%esp)
0xb36d80b1: jmp    0xb36d80c4
0xb36d80b6: push   %edx
0xb36d80b7: push   %eax
0xb36d80b8: jmp    0xb36d80c4
0xb36d80bd: push   %eax
0xb36d80be: jmp    0xb36d80c4
0xb36d80c3: push   %eax
0xb36d80c4: movsbl 0x1(%esi),%eax
0xb36d80c8: movzbl 0x2(%esi),%ebx
0xb36d80cc: add    $0x2,%esi
0xb36d80cf: iml    *-0x48f106a0(,%ebx,4)
0xb36d80d6: xchg   %ax,%ax
0xb36d80d8: add    %al,(%eax)
0xb36d80da: add    %al,(%eax)
0xb36d80dc: add    %al,(%eax)
0xb36d80de: add    %al,(%eax)
```

图 9.3 bipush 字节码指令的取指指令

图 9.3 中所框住的部分便是 bipush 字节码的取指机器码，而所框住部分之前的逻辑，是 bipush 字节码指令本身的逻辑。再如 iadd 的本地机器指令如图 9.4 所示。

```
iadd 96 iadd [0xb36d9f40, 0xb36d9f60] 32 bytes
```

```
[Disassembling for mach='i386']
0xb36d9f40: pop    %eax
0xb36d9f41: pop    %edx
0xb36d9f42: add    %edx,%eax
0xb36d9f44: movzbl 0x1(%esi),%ebx
0xb36d9f48: inc    %esi
0xb36d9f49: iml    *-0x48f106a0(,%ebx,4)
0xb36d9f50: add    %al,(%eax)
0xb36d9f52: add    %al,(%eax)
0xb36d9f54: add    %al,(%eax)
0xb36d9f56: add    %al,(%eax)
0xb36d9f58: add    %al,(%eax)
0xb36d9f5a: add    %al,(%eax)
0xb36d9f5c: add    %al,(%eax)
0xb36d9f5e: add    %al,(%eax)
```

图 9.4 iadd 字节码指令的取指指令

同样可以看到，iadd 指令本身的逻辑在前面，而取指逻辑则在后面。

其实，JVM 虚拟机的这种取指逻辑安排与硬件 CPU 的取指逻辑是完全一致的，只不过 CPU 是通过数字电路来完成自动取指功能。当 CPU 读取到当前机器指令时，CPU 内部的数字电路便会触发取指电路去工作，取指电路会根据当前所取指令的内存地址偏移当前指令的长度，从而计算出下一条指令的内存位置。通过这种机制物理 CPU 便能够周而复始地一直执行下去，直到最后一条指令。

9.2.4 程序计数器在哪里

经常可以在各种书籍上看到 JVM 在执行字节码指令时，会有一个 pc 计数器 (program counter) 指向当前所执行的指令，当前指令执行完成之后，PC 会自动指向下一条字节码指令，程序计数器是保证软件程序能够连续执行下去的关键技术之一。物理 CPU 中专门有一个寄存器用于存放 PC，当计算机上的某个软件程序开始运行之前，操作系统会将该软件程序加载至内存中（数据段、代码段），加载之后，操作系统便会执行一件非常重要的事情——将该软件程序的第一条机器指令在内存中的地址送入程序计数器，操作系统会从该地址读取指令，并开始执行，由此开始操作系统将 CPU 的控制权交给软件程序。当执行指令时，处理器将自动修改 PC 的内容，每执行一条指令，PC 便会增加一个量，这个量等于指令所含的字节数，这样 PC 所指向的内存位置总是将要执行的下一条指令的地址。由于大多数指令都是按顺序来执行的，所以 PC 通常都是加 1。

JVM 内部的程序计数器的原理也与之相同，其实经过前面对 JVM 取指技术实现的分析可知，JVM 内部的所谓 PC 计数器，其实就是 esi 寄存器（x86 平台）。当 JVM 开始执行 Java 程序的 main() 主函数时，PC（esi 寄存器）便会指向 Java 程序的 main() 主函数的第一条字节码指令的内存位置，接着 JVM 每执行完一条字节码指令便会对 PC 执行一定的增量，从而让 PC 总是指向即将要执行的字节码指令，如此便能让 Java 程序连续执行下去。

知其然，并知其所以然，方为大善！可能聪明的你会问：JVM 为何要使用宝贵的寄存器资源作为程序计数器呢？道理很简单，就是因为 CPU 读写寄存器的速度是最快的（相对于内存和磁盘）。由于 JVM 一旦跑起来之后，所做的事情就是不断地执行“取指→译码→执行”这一循环往复的任务，因此取指在 JVM 内部可谓是最频繁的事情，如此多的数据读写，如果性能低下，必然影响到 JVM 的整体执行效率，因此 JVM 便选择一个寄存器作为程序计数器。另一方面，JVM 的指令集是面向栈的，而面向栈的指令集并不直接依赖于寄存器，因此 JVM 也有更多的资源可以直接基于寄存器。后文会通过一个 Java 程序示例来演示取指的过程，以及随着 Java 程序的执行，程序计数器的变化过程。

9.3 译码

前面详细讲解了 JVM 的取指逻辑的实现机制,对于执行引擎而言,取指只是第一步,执行才是终极目标。不过从取指到执行中间,还有一个步骤:译码。

之所以要译码,道理很简单,JVM 内部定义了两百多个字节码指令,不同字节码指令的实现机制都是不同的,因此 JVM 取出字节码指令后,需要将其翻译成不同的逻辑,然后才能执行。

9.3.1 模板表

对于物理 CPU 而言,译码逻辑直接固化在硬件数字电路中,当 CPU 读取到特定的物理机器指令时,会触发所固化的特定数字电路,这种触发机制其实便是译码逻辑。如果 CPU 对一条机器指令无动于衷,那么 CPU 便无法完成译码,更无法执行指令,严重的则直接宕机。JVM 是虚拟的机器,没有专门的硬件译码电路,因此仅能软件模拟。如果 JVM 以模板解释器来解释字节码,则这种模板定义如下所示:

清单: /src/share/vm/interpreter/templateTable.cpp

功能: JVM 指令模板定义

```
void TemplateTable::initialize() {
    // ...
    // Java spec bytecodes      ubcp|disp|clvm|iswd in  out  generator      argument
    def(Bytecodes::_nop         , _|_|_|_|_|_|_|_|, vtos, vtos, nop         , _    );
    def(Bytecodes::_aconst_null, _|_|_|_|_|_|_|_|, vtos, atos, aconst_null, _    );
    def(Bytecodes::_iconst_m1   , _|_|_|_|_|_|_|_|, vtos, itos, iconst      , -1   );
    def(Bytecodes::_iconst_0    , _|_|_|_|_|_|_|_|, vtos, itos, iconst      , 0    );
    def(Bytecodes::_iconst_1    , _|_|_|_|_|_|_|_|, vtos, itos, iconst      , 1    );
    def(Bytecodes::_iconst_2    , _|_|_|_|_|_|_|_|, vtos, itos, iconst      , 2    );
    def(Bytecodes::_fload_1     , _|_|_|_|_|_|_|_|, vtos, ftos, fload       , 1    );
    def(Bytecodes::_fload_2     , _|_|_|_|_|_|_|_|, vtos, ftos, fload       , 2    );
    // ...
}
```

在该函数中,通过 def()函数对每一个字节码指令进行定义,定义了呢? def()函数一共有 9 个人参,第 1 个人参是字节码指令编码,而第 8 个人参则是该指令所对应的汇编指令生成器,这种生成器在 JVM 内部被称作 generator。其实这很好理解,因为对于模板解释器而言,每一个 Java 字节码指令最终都会生成对应的一串本地机器码,JVM 在运行期将直接执行这些机器码。因此,对于模板解释器,JVM 为每一个字节码都专门配备了一个生成器。其实所谓生成器,说白了就是一个函数而已。

例如，对于将 `int` 型局部变量从局部变量表推送至操作数栈栈顶，JVM 中共设计了多种 `iload` 字节码指令系列，例如 `iload_0`、`iload_1` 等，而这些字节码指令所对应的 `generator` 都是 `iload()` 函数。`iload()` 函数是 CPU 平台架构相关的，并且在 `templateTable.hpp` 中定义了 2 个重载函数，如下：

```
static void iload();
static void iload(int n);
```

如果字节码指令是 `iload 6`、`iload 18` 之类的，其对应的 `generator` 生成器便是 `iload()`。如果字节码指令是 `iload_0`、`iload_1` 之类的，则对应的 `generator` 生成器是 `iload(int n)`。与将自然数推送至操作数栈栈顶的机制一样，从局部变量表加载数据到操作数栈栈顶，也并非都使用同一个字节码指令。对于 `int` 型局部变量，当其 `slot` 索引号小于等于 3 时，使用 `iload_0`、`iload_1`、`iload_2` 或者 `iload_3` 这样的字节码指令。而当 `slot` 索引号超过 3 时，便会使用 `iload slot_idx` 这样的字节码指令。`iload_0`、`iload_1`、`iload_2` 或者 `iload_3` 这样的字节码指令只占 1 字节内存空间，因为没有操作数，而 `iload slot_idx` 这样的指令会占用 2 字节空间，操作码和操作数各占 1 字节，由此可知 JVM 中之所以设计 `iload_0`、`iload_1`、`iload_2` 或者 `iload_3` 这样的字节码指令，其目的仍然是为了给字节码文件瘦身。`iload_0`、`iload_1`、`iload_2` 或者 `iload_3` 字节码指令所对应的 `generator` 是 `TemplateTable::iload(int n)` 函数，其定义如下（在 32 位 x86 平台）：

清单：/src/cpu/x86/vm/templateTable_x86_32.cpp

功能：演示 x86 平台上的 `iload` 字节码指令的机器码生成函数

```
void TemplateTable::iload(int n) {
    transition(vtos, itos);
    __ movl(rax, iaddress(n));
}
```

在该函数中，通过 `__ movl(rax, iaddress(n))` 这条指令将 Java 方法栈帧的局部变量表中指定索引号的变量传送至操作数栈栈顶。关于该函数的实现机制，后文会进行分析。在这里需要关心的一个问题是：对于在 `TemplateTable::initialize()` 函数中通过 `def()` 函数所定义的各种字节码指令的模板，JVM 是如何进行保存的，以便在运行期能够据此进行模板翻译？

这个秘密就藏在 `def()` 函数中。在 `TemplateTable::initialize()` 函数中通过 `def()` 函数定义了每一个字节码指令的机器码生成器，`def` 函数实现如下：

清单：/src/share/vm/interpreter/templateTable.cpp

功能：JVM 指令模板定义函数

```
void TemplateTable::def(Bytecodes::Code code, int flags, TosState in, TosState
out, void (*gen)(int arg), int arg) {
    // should factor out these constants
    const int ubcp = 1 << Template::uses_bcp_bit;
```

```

const int disp = 1 << Template::does_dispatch_bit;
const int clvm = 1 << Template::calls_vm_bit;
const int iswd = 1 << Template::wide_bit;
// determine which table to use
bool is_wide = (flags & iswd) != 0;
Template* t = is_wide ? template_for_wide(code) : template_for(code);
t->initialize(flags, in, out, gen, arg);
}

```

在该函数中，先通过 `Template* t = is_wide ? template_for_wide(code) : template_for(code)` 从模板表中取出当前定义的字节码指令模板，接着通过 `t->initialize(flags, in, out, gen, arg)` 对字节码指令模板进行初始化，如此便将初始化好的字节码指令模板保存到模板表中。

模板表是什么？其实在 `TemplateTable::initialize()` 函数中定义的字节码指令及其生成器便保存在模板表中，其定义如下：

清单：/src/share/vm/interpreter/templateTable.hpp

功能：JVM 指令模板表

```

class TemplateTable: AllStatic {
public:
    // ...

private:
    static bool    _is_initialized; // true if TemplateTable has been initialized
    static Template _template_table [Bytecodes::number_of_codes];
    static Template _template_table_wide[Bytecodes::number_of_codes];

    static Template* template_for (Bytecodes::Code code) { Bytecodes::check
(code); return &_template_table [code]; }
    static Template* template_for_wide(Bytecodes::Code code)
{ Bytecodes::wide_check(code); return &_template_table_wide[code]; }
    // ...
}

```

在 `TemplateTable` 类中定义了 `_template_table` 数组，该数组即是模板表，模板表记录了每个字节码指令的汇编生成器（即函数）、参数及其他相关信息。该数组的元素类型是 `Template`，而数组的大小初始化为 `Bytecodes::number_of_codes`，这就是 Java 字节码指令的数量，因此 `_template_table` 数组的初始化大小就是 Java 字节码指令的数量，道理很简单，因为每一个 Java 字节码指令对应一个模板。同时 `TemplateTable` 类中定义了模板表的访问接口 `template_for (Bytecodes::Code)` 函数，其可以根据字节码指令的编号查询对应的模板，该接口在 `TemplateTable::def()` 函数中被调用，用于读取字节码指令在模板表中所对应的模板。注意，无论是模板表 `_template_table` 还是模板表的访问接口函数，都是静态的（使用 `static` 修饰），因此在

操作系统加载 `TemplateTable` 类时便会完成模板表的初始化，只不过这个时候的模板表里的元素都是空值，模板尚未构建。因此，在 `TemplateTable::def()` 函数中才需要先从模板表中取出当前模板，并进行初始化。如此一来，等到 `TemplateTable::initialize()` 函数执行完成，则字节码指令的模板表也完成构建。

那么模板在啥时候构建呢？或者换言之，`TemplateTable::initialize()` 函数在啥时候被调用呢？其实可以很容易猜到，模板表是运行期执行 Java 字节码指令时时刻都会使用到的基础数据，因此一定在 JVM 启动期间进行构建。模板表构建的整体链路如下：

- ①.java.c: `main()`
调用 `LoadJavaVM()`
- ②.java_md.c: `LoadJavaVM()`
调用 `ifn->CreateJavaVM = (CreateJavaVM_t)dlsym(libjvm, "JNI_CreateJavaVM")`
- ③.jni.cpp: `_JNI_IMPORT_OR_EXPORT_ jint JNICALL JNI_CreateJavaVM()`
调用 `result = Threads::create_vm((JavaVMInitArgs*) args, &can_try_again)`
- ④.thread.cpp: `Thread::create_vm()`
调用 `init_globals()`
- ⑤.init.cpp: `init_globals()`
调用 `interpreter_init()`
- ⑥.interpreter.cpp: `interpreter_init()`
调用 `Interpreter::initialize()`
- ⑦.templateInterpreter.cpp: `TemplateInterpreter::initialize()`
- ⑧.templateTable.cpp: `TemplateTable::initialize()`
初始化模板表

这条链路是本地调试 HotSpot 源码时的调用链路，本地调试编译好的调试版 HotSpot 时会从 `/src/share/tools/launcher/java.c` 这个 `main()` 主函数所在的文件进入，然后一路调用下来。HotSpot 启动时进入 `java.c::main()` 主函数，`main()` 主函数由操作系统调用。接着在 `LoadJavaVM()` 中调用 `JNI_CreateJavaVM()` 接口开始创建 JVM 虚拟机，需要注意的是，在 `java_md.c::LoadJavaVM()` 过程中有两处都会调用 `JNI_CreateJavaVM()` 接口，逻辑如下（POSIX 系统平台）：

清单：/src/os/posix/launcher/java_md.c

功能：创建虚拟机接口

```
jboolean LoadJavaVM(const char *jvmpath, InvocationFunctions *ifn)
{
#ifdef GAMMA
    /* JVM is directly linked with gamma launcher; no dlopen() */
    ifn->CreateJavaVM = JNI_CreateJavaVM;
    ifn->GetDefaultJavaVMInitArgs = JNI_GetDefaultJavaVMInitArgs;
    return JNI_TRUE;
#else
    // ...

```

```

    ifn->CreateJavaVM = (CreateJavaVM_t) dlsym(libjvm, "JNI_CreateJavaVM");
    // ...
#endif /* ifndef GAMMA */
}

```

在该接口中,通过判断是否定义 GAMMA 宏,分别使用两种方式调用 JNI_CreateJavaVM(),如果设置过 GAMMA 宏则直接调用该接口,否则便通过动态链接库进行调用。当在本地编译 HotSpot 源代码并生成调试版的 HotSpot 时,为了调试跟踪方便,HotSpot 提供了 gamma 启动器,通过 gamma 启动器能够直接在本地调试 HotSpot 源代码。所谓 gamma 启动器,其实就是上面链路中的入口/src/share/tools/launcher/java.c::main()函数。而对于 Java 用户而言,启动 Java 程序时所调用的命令是%JAVA_HOME%\bin\java,Java 脚本也是使用 C 语言编写的,其逻辑与 gamma 启动器的逻辑基本一样,但是这两者都共用了同一套 launcher 源代码,launcher 就是上述链路中的 java_md.c,因此需要在 java_md.c 中区分 HotSpot 是否从 gamma 启动器启动。如果不是从 gamma 启动,便是从%JAVA_HOME%\bin\java 命令中启动 HotSpot 虚拟机,如果通过 Java 命令启动虚拟机,则本地一定需要先行安装 JDK,否则 Java 程序无法启动。安装 JDK 之后,在 Linux 上会生成 libjvm.so 动态链接库,在 Windows 上会生成 jvm.dll,因此如果是从 Java 命令启动 HotSpot 虚拟机,则需要在 launcher 中加载动态链接库,从而进入虚拟机。

不管是从 gamma 启动器启动虚拟机还是从 Java 命令启动虚拟机,最终都会调用 jni.cpp::JNI_IMPORT_OR_EXPORT_jint JNICALL JNI_CreateJavaVM()接口,在该接口中会调用 Threads::create_vm()接口,该接口会执行一系列的虚拟机初始化逻辑,而 JVM 是基于解释的虚拟机,因此最终会调用 Interpreter::initialize()接口对解释器进行初始化。

在 interpreter.cpp::interpreter_init()中调用 Interpreter::initialize()时,会根据系统所设置的参数而调用对应的解释器。看 Interpreter 类的声明:

清单: /src/share/vm/interpreter/interpreter.hpp

功能: Interpreter 解释器类声明

```

class Interpreter: public CC_INTERP_ONLY(CppInterpreter) NOT_CC_INTERP
(Interpreter) {
    // ...
}

```

Interpreter 类继承时通过 CC_INTERP_ONLY(CppInterpreter)宏和 NOT_CC_INTERP(Interpreter)宏来判断继承哪一个解释器,在 HotSpot 中,默认使用模板解释器,因此 Interpreter 类实际继承的是 TemplateInterpreter 这个模板解释器。在 interpreter.cpp::interpreter_init()函数中调用 Interpreter::initialize()函数时,实际调用的是 TemplateInterpreter::initialize()这个函数,该函数声明如下:

清单：/src/share/vm/interpreter/templateInterpreter.cpp

功能：模板解释器的初始化

```
void TemplateInterpreter::initialize() {

    // 初始化解释器
    AbstractInterpreter::initialize();

    // 构建模板表
    TemplateTable::initialize();

    // 构建解释器生成器
    { ResourceMark rm;
      TraceTime timer("Interpreter generation", TraceStartupTime);
      int code_size = InterpreterCodeSize;
      NOT_PRODUCT(code_size *= 4;) // debug uses extra interpreter code space
      _code = new StubQueue(new InterpreterCodeletInterface, code_size, NULL,
                           "Interpreter");
      InterpreterGenerator g(_code);
    }

    // 初始化转发表
    _active_table = _normal_table;
}
```

在模板解释器的初始化逻辑中，主要初始化了抽象解释器 `AbstractInterpreter`、模板表 `TemplateTable`、`CodeCache` 的 Stub 队列 `StubQueue` 和解释器生成器 `InterpreterGenerator`。关于这几个逻辑在后文都会进行详细分析，在这里只需要知道，在模板解释器的初始化链路中，调用到了 `TemplateTable::initialize()` 这个接口进行模板表的构建，因此在 JVM 启动过程中，解释器初始化完成，模板表也就构建完成，每一个字节码指令都与其对应的生成器函数一一进行关联。

9.3.2 汇编器

对于模板解释器，每一个字节码指令都会关联一个生成器函数，用于生成字节码指令的本地机器码。例如 `iload_1` 字节码指令所对应的函数是 `TemplateTable::iload(int n)`，在该函数中主要调用 `__movl(rax, iaddress(n))` 函数生成对应的机器指令，所生成的机器指令是 `mov reg, operand`，表示将操作数（立即数）传送至指定的寄存器中。在 x86 平台上，该函数会调用如下接口：

清单：/src/cpu/x86/vm/assembler_86.cpp

功能：`movl()` 机器指令生成接口

```
void Assembler::movl(Register dst, Address src) {
```



```

InstructionMark im(this);
prefix(src, dst);
emit_byte(0x8B);
emit_operand(dst, src);
}

```

该函数属于 `Assembler` 类, `Assembler` 类是 JVM 内部为模板解释器所定义的汇编器, 当 JVM 使用模板解释器来解释执行字节码指令时, 便会通过汇编器来为每一个字节码指令生成对应的本地机器码。

在 `Assembler::movl()` 函数中调用 `emit` 系列的接口生成本地机器码, `emit` 系列的接口则定义在 `AbstractAssembler` 类中, 该类顾名思义是“抽象汇编器”。`emit()` 接口将机器码写入特定的内存位置, JVM 在运行期解释字节码指令时, 会跳转到特定的内存位置执行机器码。该函数会在后文讲解。

每个字节码指令都会关联一个生成器函数, 而生成器函数会调用汇编器生成机器码, 但是在 HotSpot 源码中并没有在生成器函数中看到直接调用汇编器的地方。仍以 `iload_1` 字节码为例, 其所对应的生成器函数 `TemplateTable::iload(int n)` 仅仅只是调用了 `__movl(rax, iaddress(n))`, 并没有调用类似 `assembler.movl(rax, iaddress(n))` 的函数, 然而 `movl(rax, iaddress(n))` 函数的确定义在 `/src/cpu/x86/vm/assembler_86.cpp` 这个汇编器类中 (x86 平台), 这究竟是如何关联的呢? 秘密就隐藏在 `__movl(rax, iaddress(n))` 这句指令的前缀 “`__`” 里面, 这是两个连续的下画线。事实上这是一个宏, 在 x86 平台上, 这个宏的定义如下:

清单: `/src/cpu/x86/vm/templateTable_x86_32.cpp`

功能: 模板表生成器函数中调用的汇编器宏

```

#ifndef CC_INTERP
#define __ _masm->
// ...
#endif /* !CC_INTERP */

```

因此, 在模板表中可以通过添加 `__` 前缀直接调用汇编器中的函数, 而不用添加类名。再以 `iload_1` 字节码指令为例, 在 x86 平台上其对应的生成器函数是 `iload(int n)`, 该函数调用如下函数生成本地机器码:

```
__ movl(rax, iaddress(n));
```

在 HotSpot 源码编译的预处理阶段, 这句代码会被进行宏替换, 替换之后的代码变成如下:

```
_masm->movl(rax, iaddress(n));
```

如此一来, 模板表便与汇编器关联上了, 模板表的确是通过调用汇编器接口完成本地机器码指令的生成。

不过充满求知欲的你可能接着会想，`/src/cpu/x86/vm/templateTable_x86_32.cpp` 中的 `_masm` 这个变量定义在哪里？啥时候初始化？这是一个很好的问题。对于 `TemplateTable` 类，其 `_masm` 变量定义如下：

清单：`/src/share/vm/interpreter/templateTable.hpp`

功能：模板表中汇编器变量定义

```
class TemplateTable: AllStatic {
public:
    // ...
    static InterpreterMacroAssembler* _masm; //汇编器

private:
    // ...
}
```

注意：`TemplateTable` 模板表中的汇编器变量类型是静态的。汇编器变量在 JVM 启动期间，JVM 调用字节码指令所对应的生成器函数时会对其进行赋值。在这里，不得不再次提起 JVM 的取指逻辑，上文在讲解取指逻辑时对其进行过详细分析。在 JVM 启动期间，JVM 会为所有字节码指令生成取指逻辑，HotSpot 通过 `TemplateInterpreterGenerator::generate_and_dispatch()` 接口来生成取指逻辑，上文讲过，HotSpot 为每一个字节码指令生成“取指”逻辑的同时（其实是指取下一条字节码指令），会为该字节码生成其本身所对应的机器逻辑指令。在 `TemplateInterpreterGenerator::generate_and_dispatch()` 接口中，会先调用 `t->generate(_masm)` 函数，为当前字节码指令生成字节码本身的机器逻辑，接着才会调用 `dispatch_epilog(tos_out, step)` 函数为该字节码指令生成其对应的取指逻辑（取下一条字节码指令）。注意观察，在调用 `t->generate(_masm)` 时，为其传递了一个 `_masm` 指针，这个指针也是汇编器，其实模板表的静态变量 `_masm` 函数便是在 JVM 调用 `t->generate(_masm)` 函数时进行了赋值，看 `t->generate(_masm)` 函数定义：

清单：`/src/share/vm/interpreter/templateTable.cpp`

功能：模板解释器的汇编器赋值

```
void Template::generate(InterpreterMacroAssembler* masm) {
    // parameter passing
    TemplateTable::_desc = this;
    TemplateTable::_masm = masm;
    // code generation
    _gen(_arg);
    masm->flush();
}
```

在调用该函数时会将汇编器作为参数传递进来,而在该函数中调用`_gen(_arg)`函数时,由于`_gen`是各个字节码指令所对应的本地机器码生成函数,这些生成函数都是`TemplateTable`类的静态函数,因此JVM在调用这些函数时,这些生成器函数会调用汇编器的接口生成本地机器码,生成器函数所调用的汇编器实际上便是`Template::generate()`函数所传入的汇编器。

有点绕不是?还是以`iload_1`字节码指令为例来理一理思路,首先,JVM启动期间调用`TemplateTable`类的`initialize()`接口将每一个字节码指令与其生成器函数进行关联,例如`iload_1`字节码指令所关联的生成器函数就是`TemplateTable::iload(int i)`,但是在这个阶段,`TemplateTable::iload(int i)`并不会被调用。接着,JVM启动期间会调用模板解释器(`templateInterpreter`)为每个字节码指令生成该字节码指令的本地机器码指令(这个后文会讲解),同时会为该字节码生成对应的取指的本地机器码指令(取下一条字节码指令),这两个逻辑都封装在`TemplateInterpreterGenerator::generate_and_dispatch()`接口中,在该接口中调用`t->generate(_masm)`函数来为当前字节码指令生成机器指令,注意,模板表`TemplateTable::_masm`静态变量便在这个函数内部完成初始化,在`t->generate(_masm)`函数内部执行`TemplateTable::_masm = masm`将外部所传递进来的汇编器实例传递进`TemplateTable`模板表内部。`t->generate(_masm)`内部调用`_gen(_arg)`函数为字节码指令生成本地机器码。由于`iload_1`字节码指令对应的`_gen`是`TemplateTable::iload(int i)`函数,因此JVM会调用`TemplateTable::iload(int i)`函数。在`TemplateTable::iload(int i)`函数内部调用`__movl(rax, iaddress(n))`生成本地机器码。注意,这里的前缀“__”是两条下画线,这是一个宏,前面刚讲过,这个宏会在预处理阶段被替换成`_masm->`,因此JVM实际上调用的是`TemplateTable::_masm->movl(rax, iaddress(n))`函数。由于刚才在执行`t->generate(_masm)`函数时,JVM根据该函数所传入的汇编器`_masm`对`TemplateTable::_masm`变量进行了初始化,因此JVM所调用的便是传递给`t->generate(_masm)`函数的汇编器的`movl()`接口。其主体链路如下:

①.jvm启动

```
...
调用 TemplateTable::initialize()
```

②.TemplateTable::initialize()

③.TemplateTable::def()

```
iload_1 字节码指令的 _gen 生成器映射为 TemplateTable::iload(int i) 函数
```

④.TemplateInterpreterGenerator::generate_and_dispatch()

⑤.t->generate(_masm)

```
这里用 _masm 实例对 TemplateTable::_masm 进行了初始化
```

⑥._gen(_arg)

```
这里实际调用的是 TemplateTable::iload(int i)
```

⑦.__movl(rax, iaddress(n))

```
这里实际调用的是 TemplateTable::_masm->movl()
```

对照上面的文字看这条链路，思路应该会清晰很多。至此，关于 TemplateTable 模板表中的汇编器 _masm 什么时候被初始化的问题理出了头绪，相信聪明的你立马会想到：既然 TemplateTable 模板表中的汇编器是在 JVM 调用 t->generate(_masm) 函数时被初始化，那么这里所传入的 _masm 汇编器又是啥呢？其实也很简单，首先确定这里的 _masm 指针所声明的类。由于 t->generate(_masm) 函数在 TemplateInterpreterGenerator::generate_and_dispatch() 函数中被调用，因此可以确定 TemplateInterpreterGenerator::generate_and_dispatch() 函数里的 _masm 变量是 TemplateInterpreterGenerator 类中的变量，TemplateInterpreterGenerator 是模板解释器生成器，所谓解释器生成器，前面讲过，是专门负责为模板解释器将字节码指令翻译生成本地机器码的类型。HotSpot 内部有好几种解释器，其他解释器也有自己专门的生成器，负责将字节码指令解释为特定的逻辑。TemplateInterpreterGenerator 继承自 AbstractInterpreterGenerator 类（看 /src/share/vm/interpreter/abstractInterpreter.hpp 文件），AbstractInterpreterGenerator 中便定义了一个汇编器指针，如下：

清单：/src/share/vm/interpreter/abstractInterpreter.hpp

功能：解释器生成器中的汇编器变量声明

```
class AbstractInterpreterGenerator: public StackObj {
protected:
    InterpreterMacroAssembler* _masm;
    // ...
}
```

注意，这个类名是 AbstractInterpreterGenerator，顾名思义，该类是“抽象解释器生成器”，是解释器生成器的顶级父类。既然模板解释器生成器 TemplateInterpreterGenerator 继承自这个基类，那么 TemplateInterpreterGenerator 自然也继承了汇编器指针 _masm，所以在 TemplateInterpreterGenerator::generate_and_dispatch() 函数中 JVM 才可以将汇编器指针直接传入 Template::generate() 函数中。

至此，关于汇编器初始化的问题，便演化成 TemplateInterpreterGenerator 中的汇编器啥时候初始化的问题。其实，汇编器初始化的逻辑隐藏在 CodeletMark 类的构造函数中，该类的定义如下：

清单：/src/share/vm/interpreter/interpreter.hpp

功能：CodeletMark 类的构造函数

```
class CodeletMark: ResourceMark {
private:
    InterpreterCodelet* _clet; // 解释器脚本代码
    InterpreterMacroAssembler** _masm; // 这里定义了汇编器
    CodeBuffer _cb; // 代码缓存
```

```

// ...

public:
    CodeletMark(
        InterpreterMacroAssembler*& masm,
        const char* description,
        Bytecodes::Code bytecode = Bytecodes::_illegal):
        _clet((InterpreterCodelet*)AbstractInterpreter::code()->
request(codelet_size())),
        _cb(_clet->code_begin(), _clet->code_size())
    {
        // initialize Codelet attributes
        _clet->initialize(description, bytecode);

        // 实例化一个汇编器
        masm = new InterpreterMacroAssembler(& _cb);
        _masm = &masm; //将外部传进来的汇编器指针指向刚刚创建的汇编器实例对象
    }

    // ...
};

```

纵观整个 HotSpot 源码，只有在 CodeletMark 类的构造函数里对汇编器进行了实例化。CodeletMark 其实是一个包装器，能够自动回收所分配的代码空间以及所实例化的汇编器。那么 CodeletMark 类与 TemplateInterpreterGenerator 类中的汇编器有啥关系呢？说到这里，就不得不先讲一讲 HotSpot 字节码指令生成本地机器码的流程了。默认情况下 HotSpot 会启用模板解释器来解释执行字节码指令，在 JVM 启动期间，JVM 会依次调用模板解释器的生成器为各个字节码指令生成对应的本地机器指令及各个字节码指令的取指逻辑，前文讲过，这个逻辑主要封装在 TemplateInterpreterGenerator::generate_and_dispatch()这个函数中。该函数调用的整体链路如下：

- ①.java.c: main() 调用 LoadJavaVM()
 - ②. java_md.c: LoadJavaVM() 调用 ifn->CreateJavaVM = (CreateJavaVM_t)dlsym(libjvm, "JNI_CreateJavaVM")
 - ③.jni.cpp: JNI_IMPORT_OR_EXPORT jint JNICALL JNI_CreateJavaVM() 调用 result = Threads::create_vm((JavaVMInitArgs*) args, &can_try_again)
 - ④.thread.cpp: Thread::create_vm() 调用 init_globals()
 - ⑤.init.cpp: init_globals() 调用 interpreter_init()
 - ⑥.interpreter.cpp: interpreter_init() 调用 Interpreter::initialize()
 - ⑦.templateInterpreter.cpp: TemplateInterpreter::initialize()
 - ⑧.templateTable.cpp: TemplateTable::initialize() 初始化模板表
 - ⑨.templateInterpreter_x86_32.cpp: InterpreterGenerator(_code) 解释器生成器构造函数
- templateInterpreter_x86_32.cpp:

```

TemplateInterpreterGenerator::generate_all()
⑩.templateInterpreter.cpp:
TemplateInterpreterGenerator::set_entry_points_for_all_bytes()
⑪.templateInterpreter.cpp:
TemplateInterpreterGenerator::set_entry_points()
⑫.templateInterpreter.cpp:
TemplateInterpreterGenerator::set_short_entry_points()
⑬.templateInterpreter.cpp:
TemplateInterpreterGenerator::generate_and_dispatch()

```

上面这条链路是在 32 位 x86 平台上执行时的路径。其中与汇编器有关的关键点便在 TemplateInterpreterGenerator::set_entry_points() 这个分支流中。前面多次提到，JVM 在启动期间会调用 TemplateInterpreterGenerator::generate_and_dispatch() 接口为每个字节码指令生成对应的本地机器码及对应的取指指令，通过上面这条链路可以很清晰地看出来，TemplateInterpreterGenerator::generate_and_dispatch() 接口便是由 TemplateInterpreterGenerator::set_entry_points() 函数所调用，而该函数本身则又是由 TemplateInterpreterGenerator::set_entry_points_for_all_bytes() 函数所调用。这两个函数的主要逻辑如下：

清单：/src/share/vm/interpreter/templateInterpreter.cpp

功能：set_entry_points() 与 set_entry_points_for_all_bytes 函数

```

// 为所有字节码指令设置入口点
// DispatchTable::length 的值就是字节码指令集的总数量
void TemplateInterpreterGenerator::set_entry_points_for_all_bytes() {
    // 遍历所有字节码指令
    for (int i = 0; i < DispatchTable::length; i++) {
        Bytecodes::Code code = (Bytecodes::Code)i;
        if (Bytecodes::is_defined(code)) {
            set_entry_points(code);
        } else {
            set_unimplemented(i);
        }
    }
}

// 为指定字节码指令设置全部入口点
void TemplateInterpreterGenerator::set_entry_points(Bytecodes::Code code) {
    CodeletMark cm(_masm, Bytecodes::name(code), code);
    // ...
    if (Bytecodes::is_defined(code)) {
        Template* t = TemplateTable::template_for(code);
        set_short_entry_points(t, bep, cep, sep, aep, iep, lep, fep, dep, vep);
    }
    if (Bytecodes::wide_is_defined(code)) {

```

```

Template* t = TemplateTable::template_for_wide(code);
set_wide_entry_point(t, wep);
}

// set entry points
EntryPoint entry(bep, cep, sep, aep, iep, lep, fep, dep, vep);
Interpreter::_normal_table.set_entry(code, entry);
Interpreter::_wentry_point[code] = wep;
}

```

TemplateInterpreterGenerator::set_entry_points_for_all_bytes()函数的逻辑很好理解,遍历每一个字节码,然后调用 TemplateInterpreterGenerator::set_entry_points()函数为各个字节码指令设置入口点。关于入口点的概念在后文会分析,这里将目光聚焦于 TemplateInterpreterGenerator::set_entry_points()函数,该函数一开始便通过 CodeletMark cm(_masm, Bytecodes::name(code), code)实例化了一个 CodeletMark 类对象,这里就是关键点。实例化 CodeletMark 时,向其构造函数传递了指针变量_masm,该指针便是 TemplateInterpreterGenerator 类从抽象解释器生成器 AbstractInterpreterGenerator 中所继承而来的私有成员变量,该指针指向汇编器的内存首地址。在通过 CodeletMark cm(_masm, Bytecodes::name(code), code)创建 CodeletMark 类型实例对象时,TemplateInterpreterGenerator 类中的私有成员变量_masm 尚未初始化,但是上文刚刚讲到,在 CodeletMark 的构造函数中会实例化一个汇编器,并将外部所传入的汇编器指针指向这个所创建的汇编器实例对象。如此一来,当 TemplateInterpreterGenerator::set_entry_points()函数执行完 CodeletMark cm(_masm, Bytecodes::name(code), code)之后,TemplateInterpreterGenerator 类的私有成员变量_masm 便完成初始化,至此,本节一直在研究的问题——汇编器何时初始化,总算有了答案。

当 TemplateInterpreterGenerator 类的私有成员变量_masm 完成初始化之后,则当调用到上面的链路图中的最后链路 TemplateInterpreterGenerator::generate_and_dispatch()时,JVM 便将_masm 汇编器指针通过 t->generate(_masm)调用传递给了 TemplateTable::_masm 这个静态字段,最终在 TemplateTable 模板表中调用相关函数为字节码指令生成本地机器码时,实际所调用的便是 TemplateTable::_masm 这个静态字段所指向的汇编器实例的接口。那么接下来,聪明的你可能又会禁不住发问,汇编器到底是啥?到底是怎么将字节码指令翻译成对应的机器码呢?

在 CodeletMark 的构造函数中,直接将_masm 实例化成 InterpreterMacroAssembler 这个汇编器类型。事实上,在 HotSpot 内部,汇编器总共包含 4 个继承层次,如图 9.5 所示(32 位 x86 平台)。

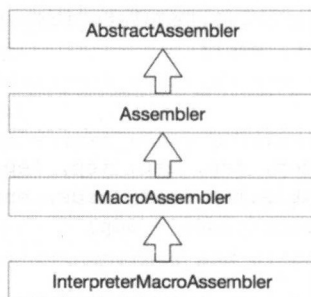


图 9.5 HotSpot 在 x86 平台上的 4 层汇编器继承关系

这 4 层汇编器所对应的文件分别如下：

- ◎ AbstractAssembler, assembler.hpp
- ◎ Assembler, assembler_x86.hpp
- ◎ MacroAssembler, assembler_x86.hpp
- ◎ InterpreterMacroAssembler, interp_masm_x86_32.hpp

顶级汇编器是 AbstractAssembler，顾名思义，就是抽象汇编器，但是该抽象汇编器其实一点都不抽象，它定义了最核心的功能和数据结构，如下：

清单：/src/share/vm/asm/assembler.hpp

功能：抽象汇编器的接口和核心字段

```

class AbstractAssembler : public ResourceObj {

protected:
    address    _code_begin;           // 指令缓存区首地址
    address    _code_pos;             // 写入的当前位置

    // ...

    void emit_byte(int x); // 往指令区写入一个字节数据/指令
    void emit_word(int x); // 往指令区写入一个 16 位的数据
    void emit_long(jint x); // 往指令区写入一个 32 位的数据
    void emit_address(address x); // 往指令区写入一个地址数据
    // ...
}
    
```

每一个汇编器都会往内存中写入一段指令，因此 JVM 会为每一个汇编器分配一个内存首地址，汇编器就从该首地址开始写入指令或数据。因此在这个抽象汇编器中定义了 `_code_begin`、`_code_pos` 等字段，用于记录首地址及当前写入的位置。同时，抽象汇编器提供了写入本地机器指令的接口，就是 `emit()` 系列的函数。可以写入一个 8 位、16 位或者其他宽度的数据/指令。抽

象汇编器的子类都依赖于这些核心功能生成机器指令。而在汇编器的继承体系中，继承层次越深的汇编器，其处理的业务越抽象和复杂，也与 Java 的字节码指令越靠近。同时，除了最顶级的抽象汇编器，其子类皆与具体的硬件平台相关，这很好理解，因为机器码本身就是硬件相关的。对于 `Assembler` 和 `MacroAssembler`，在各种平台相关的源码中都有定义，例如在 `assembler_sparc.hpp` 中也定义了这两个类。

如果想深入理解 HotSpot 的解释执行原理，就必须对这几个汇编器子类有深入理解，下面对其进行简单的分析。

9.3.3 汇编

HotSpot 内部定义了 4 个层次的汇编器，顶级的抽象汇编器提供了往缓冲区写入本地机器码指令和数据的接口，并记录所写入的起始地址与当前地址。例如，`emit_byte()` 接口提供往指令缓冲区写入一个字节码宽度的指令/数据的能力，其实现如下：

清单：/src/share/vm/asm/assembler.inline.hpp

功能：汇编器生成 1 字节的机器码指令功能演示

```
inline void AbstractAssembler::emit_byte(int x) {
    assert(isByte(x), "not a byte");
    *(unsigned char*)_code_pos = (unsigned char)x;
    _code_pos += sizeof(unsigned char);
    sync();
}
```

该函数在 `_code_pos` 所指向的内存位置处写入一个输入的字节指令或数据，写入后，将 `_code_pos` 指针再往前移动 1 字节，等待写入下一个机器指令。hotspot 默认使用模板解释器，JVM 在启动期间会初始化模板解释器，为每一个字节码指令在内存中写入其对应的机器指令片段，JVM 在运行期便能够对字节码指令进行解释，当执行某一条字节码指令时，会读取其对应的机器指令并执行，从而完成 Java 逻辑运算。

位于汇编器继承体系第二层的是 `Assembler` 汇编器。`Assembler` 其实是对物理机器指令的抽象，或者说软件封装。例如，在 x86 平台上，`Assembler` 类中的接口如下：

清单：/src/cpu/x86/vm/assembler_x86.hpp

功能：x86 平台上的 `Assembler` 定义

```
class Assembler : public AbstractAssembler {
    // ...

    void decl(Register dst);
    void decl(Address dst);
}
```

```

void incl(Register dst);
void incl(Address dst);

void lea(Register dst, Address src);
void mov(Register dst, Register src);

void push(int32_t imm32);
void push(Register src);
void pop(Register dst);

// ...
}

```

如果你熟悉汇编指令,应该对上面 Assembler 类中的这些 dec()、inc()、lea()和 mov()等接口感到非常亲切,例如, x86 平台提供的压栈指令中有一种指令格式是 push reg, 于是 Assembler 类中便提供了一个接口 push(Register src), 这表示将一个硬件寄存器的值压入栈中。另一种压栈指令格式是 push operand, 这表示将一个立即数压入栈中, 于是 Assembler 类中便提供了一个接口 push(int32_t imm32)。Assembler 类中的这些接口基本是对物理机器指令的“纯净”模拟, 即最终所生成出来的机器码与原生的机器指令是完全一致的, HotSpot 并不会往里面加入别的“杂项”。例如以压栈指令为例, 在 x86 平台上, 将一个立即数压栈的接口实现如下:

清单: /src/cpu/x86/vm/assembler_x86.cpp

功能: x86 平台上 Assembler::push()的实现

```

void Assembler::push(int32_t imm32) {
    emit_byte(0x68);
    emit_long(imm32);
}

```

push()函数里面调用 emit()系列的接口往指令缓存区先写入 0x68 这个字节, 再将 imm32 这个 32 位宽的立即数写入指令缓存区。emit()系列的接口前文讲过, 就是顶级抽象汇编器所提供的核心接口。0x68 是 Intel 处理器所提供的 push 机器指令的十六进制编码, 例如 push 2, 则对应的机器码是 0x68 02。因此, Assembler::push(int32_t imm32)这个接口最终会向指令缓存区中写入下面这条机器码:

```
push imm32
```

这与实际的机器指令是一致的。因此, 在 HotSpot 内部, 调用 Assembler::push(int32_t imm32) 接口, 可以完全等同于调用物理机器的 push operand 这个机器指令。所以 Assembler 类基本就是对物理机器指令的完全抽象, 并且是“纯净版”的软抽象。

在汇编器继承体系中, 到了 Assembler 汇编器的下一层, 便是 MacroAssembler 汇编器, 这

一层的汇编器仍然基于机器硬件指令进行抽象，但是不再是“纯净版”的抽象和模拟，而是被打上了深深的 Java 烙印，很多指令能够直接为 Java 数据所用。看看 MacroAssembler 在 x86 平台上的定义：

清单：/src/cpu/x86/vm/assembler_x86.hpp

功能：x86 平台上的 MacroAssembler 定义

```
class MacroAssembler: public Assembler {

    // bool 类型数据传送
    void movbool(Register dst, Address src);
    void movbool(Address dst, bool boolconst);
    void movbool(Address dst, Register src);

    // 加载/存储 Java 类的元数据 klass
    void load_klass(Register dst, Register src);
    void store_klass(Register dst, Register src);

    // 从 JVM 堆内存中加载 Java 对象实例
    void load_heap_oop(Register dst, Address src);
    void load_heap_oop_not_null(Register dst, Address src);
    void store_heap_oop(Address dst, Register src);

    // 对内存地址和自然数进行求和
    void addptr(Address dst, int32_t src) { LP64_ONLY(addq(dst, src))
NOT_LP64(addl(dst, src)) ; }

    // 传送 Java object 对象
    void movoop(Register dst, jobject obj);
    void movoop(Address dst, jobject obj);

    // 传送数组
    void movptr(ArrayAddress dst, Register src);

    // ...
}
```

注意观察 MacroAssembler 类所提供的接口，可以发现，也有类似物理机器级别的 mov、add 等指令，但是 MacroAssembler 类更进一步，延伸出 movbool 这种传送布尔值的接口，同时能够从 JVM 堆内存中读取 Java 类实例对象。再如，在物理机器级别，求和指令格式是 add opnd1, opnd2，但是 CPU 对 add 后面的 2 个操作数有要求，opnd1 和 opnd2 均为寄存器是允许的，一个为寄存器而另一个为存储器也是允许的，但不允许两个都是存储器操作数，即不允许两个操作数都来自内存，或者一个来自内存一个来自立即数。但是在 MacroAssembler 汇编器中提供了接口 void addptr(Address dst, int32_t src)，该接口想实现的目标是将一个存储器操作数和一个立即

数累加求和，由于物理机器不支持这种指令，因此 MacroAssembler 汇编器只能对这个接口进行复杂处理，例如，在 32 位 x86 平台上，该接口最终调用如下函数才能实现累加：

清单：/src/cpu/x86/vm/assembler_x86.cpp

功能：MacroAssembler 类中实现将存储单元与立即数累加求和的逻辑

```
void Assembler::addl(Address dst, int32_t imm32) {
    InstructionMark im(this);
    prefix(dst);
    emit_arith_operand(0x81, rax, dst, imm32); // 0x81 是 Intel 平台上的 add 指令的十六进制编码
}

// immediate-to-memory forms
void Assembler::emit_arith_operand(int opl, Register rm, Address adr, int32_t imm32) {
    if (is8bit(imm32)) {
        emit_byte(opl | 0x02); // set sign bit
        emit_operand(rm, adr, 1);
        emit_byte(imm32 & 0xFF);
    } else {
        emit_byte(opl);
        emit_operand(rm, adr, 4);
        emit_long(imm32);
    }
}
```

可以看到，对于这种物理机器不支持的指令，JVM 内部需要生成多条机器指令去处理，才能完成 MacroAssembler 汇编器中所定义的一个接口功能。因此，MacroAssembler 汇编器可以被看作对物理机器指令的组合封装（也可以认为是对其父类汇编器 Assembler 的组合封装），同时 MacroAssembler 能够支持 Java 内部数据对象级别的机器指令原语操作，从而为 JVM 内部的解释器完成特定逻辑提供必要的支撑。关于 MacroAssembler 类中的其他接口，各位有兴趣的道友可以自行深入研究。总之，汇编器模块应该代表了整个虚拟机中最精华的部分，而能否将精华全部消化吸收，完全就看个人的能力和修为。不过对于并非从事 JVM 开发的道友而言，倒没必要面面俱到，只要领悟了其思想便可。

在汇编器的继承体系中，MacroAssembler 汇编器的下一层是 InterpreterMacroAssembler 汇编器，顾名思义，这是解释器级别的汇编器，其直接为解释器提供相关汇编接口。先来看看这个类中都定义了些什么指令接口：

清单：/src/cpu/x86/vm/interp_masm_x86_32.cpp

功能：InterpreterMacroAssembler 汇编器主要接口

```
class InterpreterMacroAssembler: public MacroAssembler {
```

```

// ...

// 运行时获取参数或相关结果
void get_method(Register reg) { movptr(reg,
Address(rbp, frame::interpreter_frame_method_offset * wordSize)); }
void get_constant_pool(Register reg) { get_method(reg);
movptr(reg, Address(reg, methodOopDesc::constants_offset())); }
void get_constant_pool_cache(Register reg)
{ get_constant_pool(reg); movptr(reg, Address(reg,
constantPoolOopDesc::cache_offset_in_bytes())); }
void get_cpool_and_tags(Register cpool, Register tags)
{ get_constant_pool(cpool); movptr(tags, Address(cpool,
constantPoolOopDesc::tags_offset_in_bytes()));
}
void get_unsigned_2_byte_index_at_bcp(Register reg, int bcp_offset);
void get_cache_and_index_at_bcp(Register cache, Register index, int
bcp_offset, size_t index_size = sizeof(u2));
void get_cache_entry_pointer_at_bcp(Register cache, Register tmp, int
bcp_offset, size_t index_size = sizeof(u2));
void get_cache_index_at_bcp(Register index, int bcp_offset, size_t
index_size = sizeof(u2));

// 操作数栈相关指令
void f2ieee(); // truncate ftos to 32bits
void d2ieee(); // truncate dtos to 64bits

void pop_ptr(Register r = rax);
void pop_i(Register r = rax);
void pop_l(Register lo = rax, Register hi = rdx);
void pop_f();
void pop_d();

void push_ptr(Register r = rax);
void push_i(Register r = rax);
void push_l(Register lo = rax, Register hi = rdx);
void push_d(Register r = rax);
void push_f();

// ...

// 取指相关的指令
void dispatch_prolog(TosState state, int step = 0);
void dispatch_epilog(TosState state, int step = 0);
void dispatch_only(TosState state); // dispatch via rbx,
(assume rbx, is loaded already)
void dispatch_only_normal(TosState state); // dispatch normal
table via rbx, (assume rbx, is loaded already)

```

```

    void dispatch_only_noverify(TosState state);
    void dispatch_next(TosState state, int step = 0);           // load rbx, from
[esi + step] and dispatch via rbx,
    void dispatch_via (TosState state, address* table);         // load rbx, from
[esi] and dispatch via rbx, and table

    // jump to an invoked target
    void prepare_to_jump_from_interpreted();

    // ...
}

```

这个类中的接口分类比较明确，主要分为获取运行时参数相关的指令、操作数栈相关的指令、取指相关的指令等。事实上，还有性能监控的指令。由于 HotSpot 是一款基于栈式指令集的虚拟机，因此在 InterpreterMacroAssembler 类中可以看到各种 pop 和 push 指令接口，例如，将不同类型的数据压栈的指令包括 push_ptr()、push_i() 和 push_l() 等。同时，对于任何一个执行引擎而言，取指指令都是必须支持的（物理机器事实上没有软件取指指令，直接通过硬件数字电路触发，好高大上的感觉有没有……），因此 InterpreterMacroAssembler 类内部定义了各种取指相关的接口，这些接口统一以 dispatch 为前缀。其实，在 JVM 内部，取指也可以叫作分发，dispatch 的直接含义，这有其特殊的技术原因，下文会讲到。同时，JVM 内部在调用方法之前会创建栈帧，在动态绑定时会进行运行期链接，这些操作都需要解释器能够在运行期读取各种参数，例如，Java 方法在 JVM 内部所对应的 method 对象实例、Java class 字节码文件常量池在 Jvm 内存中的映像、字节码所对应的入口点缓存等，所以在 InterpreterMacroAssembler 类中提供了 get_method()、get_constant_pool() 等接口。以上这些核心的接口，将支撑起 JVM 内部解释器的运行期的各种调用，使得解释器能够站在 JVM 虚拟机这个层面或者这个高度来“看待”问题，或者说能够以高度抽象的思维来“思考”问题，而不仅仅局限于物理机器指令的各种原子化的琐碎的指令逻辑上。其实这本质上也是一种“面向对象”的抽象思维方式，一层一层地抽象，抽象通过软件函数的封装得以体现，函数所封装的不仅仅是函数，是能力。从机器到汇编，从汇编到 B 语言，从 B 到 C 语言，从 C 到 C++，从 C++ 到 Java，其实就是在一路抽象，一路封装，汇编是机器指令的简单符号代替，而 C 语言中的一个接口便封装了无数汇编能力，一个 Java 接口其实也封装了若干 C 和机器指令的特性。不仅编程语言如是，网络架构、分布式集群、中间件等无不是种种特性的抽象和封装。从这个角度去分析应用系统，也不难看出应用系统要分层的原因了。

大道至简。

在理解了这 4 层汇编器之后，再回头看 Java 的字节码指令。在上面所讲的这 4 层汇编器里，似乎并没有看到对 Java 字节码指令的汇编。事实上，前文讲过，字节码指令所对应的汇编生成

器都在 TemplateTable 模板表中, 模板表为每个 Java 字节码指令提供了本地机器码生成器接口 (或者解释入口), 在生成器接口中调用汇编器生成字节码指令的本地机器逻辑。仍以 `iload_1` 这条字节码指令为例, 其对应的生成器接口是 `TemplateTable::iload(int i)`, 该接口仅通过调用 `__movl(rax, iaddress(n))` 来生成机器码, 前面分析过, 这条指令会在预处理阶段将宏替换为 `_masm->movl(rax, iaddress(n))`, `_masm` 便是汇编器, `movl(rax, iaddress(n))` 函数实现在前面已经给出过, 这里不重复贴了, 该函数最终会生成如下机器码 (32 位 x86 平台):

```
mov    -0x4(%edi), %eax
```

在前面讲解 Java 方法堆栈的章节中讲过, 当 JVM 调用一个 Java 函数之前会为其创建栈帧空间, 被调用 Java 方法的局部变量表也会同时被创建, 创建完栈帧之后, JVM 便会将 `edi` 寄存器指向局部变量表的第 0 个 slot 位置。因此上面这条机器指令的含义是: 取相对于局部变量表第 0 个 slot 偏移量为 4 字节位置处 (即第一个 int 型局部变量) 的变量, 将其值传送到 `eax` 寄存器中。

如果局部变量表的 slot 索引号大于 3, 则将该变量推送至操作数栈栈顶的字节码指令便是 `iload`, 而不是 `iload_0`、`iload_1`、`iload_2` 及 `iload_3`。例如, 下面这个 Java 示例程序:

清单: /Test.java

功能: 演示 `iload` 字节码指令

```
class Test{
    public static int add(int x, int y, int z){
        int sum = x + y + z;
        int sum2 = sum + 3;
        return sum2;
    }
}
```

在 `Test` 类中定义了一个静态方法 `add()`, 其包含 3 个入参, 方法内部包含两个局部变量。由于是静态方法, 因此没隐式的 `this` 入参, 所以 3 个入参的 slot 索引号分别是 0、1、2, 而内部的两个局部变量的 slot 索引号则分别是 3 和 4。`add()` 方法最终要将第 2 个局部变量 `sum2` 的值返回出去, 因此必然会涉及一次 `iload` 操作。同时由于 `sum2` 在局部变量表中的 slot 索引号是 4, 大于 3, 因此将其推送至操作数栈栈顶的字节码指令一定是 `iload 4`, 而 `add()` 方法的入参和其内部局部变量 `sum` 的入栈指令则分别是 `iload_0`、`iload_1`、`iload_2` 和 `iload_3`。可以使用 `javap` 命令进行验证, `javap` 的输出结果如下:

```
public static int add(int, int, int);
  descriptor: (III)I
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=5, args_size=3
```



```

0: iload_0
1: iload_1
2: iadd
3: iload_2
4: iadd
5: istore_3
6: iload_3
7: iconst_3
8: iadd
9: istore      4
11: iload      4

```

从输出结果可以看出, 编译器实际所生成的 load 系列指令与所推论的完全一致。对于 iload 4 这样的字节码指令, 在模板表 TemplateTable 中为其所绑定的本地机器生成器函数是 TemplateTable::iload(), 其定义如下 (32 位 x86 平台):

清单: /src/cpu/x86/vm/templateTable_x86_32.cpp

功能: TemplateTable::iload()函数定义

```

void TemplateTable::iload() {
    transition(vtos, itos);
    if (RewriteFrequentPairs) {
        Label rewrite, done;

        // ...

        __ bind(rewrite);
        patch_bytecode(Bytecodes::_iload, rcx, rbx, false);
        __ bind(done);
    }

    // Get the local value into tos
    locals_index(rbx); //将局部变量的 slot 索引号读取到 rbx 寄存器中
    __ movl(rax, iaddress(rbx));
}

```

忽略该函数中的 if (RewriteFrequentPairs){} 分支, 该函数的主要逻辑便只剩下 locals_index(rbx)和__ movl(rax, iaddress(rbx)), 其中 locals_index(rbx)顾名思义就是将局部变量的 slot 索引号读取到 rbx 寄存器中, 而__ movl(rax, iaddress(rbx))则将指定 slot 索引号的局部变量读取到 rax 寄存器中。最终, 所生成的 TemplateTable::iload()的本地机器逻辑如下:

```

--取字节码指令的操作数
--esi 寄存器指向当前字节码指令的起始地址
--假设字节码指令是 iload 6, 则这条机器码将读取 iload 这个字节码指令所在内存位置的下一个
字节, 下一个字节的值是操作数 6
movzbl 0x1(%esi), %ebx

```

```
--求补操作
neg    %ebx
```

```
--edi 指向局部变量表第 0 个 slot 位置
```

--上一步将 `iload 6` 的操作数 6 读进了 `ebx` 寄存器中，因此这里对第 0 个 slot 偏移 6 个位置，将该位置的局部变量值读进 `eax` 寄存器中

```
mov    (%edi,%ebx,4),%eax
```

如果对汇编语法很熟悉，则很容易读懂这几行汇编指令的逻辑，这便是 `iload` 字节码指令的机器实现逻辑。

上面分别演示了 `iload_1` 和 `iload` 这 2 条字节码指令的本地机器实现，虽然不能因此而言尽 JVM 执行引擎的全部细节，但是足以管中窥豹，把握 JVM 执行引擎的整体脉络，一通而百通。而事实上，HotSpot 为了提升性能而设计了模板解释器这种直接生成本地机器指令的机制，但是在一开始可没这么高大上，一开始的解释器真的只是解释器，并不是通过模板解释器将字节码指令直接翻译成对应的机器码去执行，而是直接使用 C 语言逻辑去解释字节码指令。在 HotSpot 中至今仍然保留着最原始的字节码解释器，其对应的文件是 `bytecodeInterpreter.cpp`。

下面仍以 `iload` 字节码为例，其在 `bytecodeInterpreter` 中的逻辑如下：

清单：/src/share/vm/interpreter/bytecodeInterpreter.cpp

功能：字节码解释器中的 `iload` 指令解析

```
BytecodeInterpreter::run(interpreterState istate) {
    // ...
    switch (opcode){
        // ...

        CASE(_iload):
        CASE(_fload):
            SET_STACK_SLOT(LOCALS_SLOT(pc[1]), 0); //将局部变量推送至操作数栈栈顶
            UPDATE_PC_AND_TOS_AND_CONTINUE(2, 1); //更新 PC 计数器并取下一条字节码指令

        // ...
    }
    // ...
}
```

从这里可以看到，字节码解释器在解释 `iload` 指令时，调用 `SET_STACK_SLOT(LOCALS_SLOT(pc[1]), 0)` 函数将局部变量推送至操作数栈栈顶。该函数其实是一个宏，同时其里面的人参函数 `LOCALS_SLOT` 也是一个宏，这 2 个宏的定义如下（x86 平台）：

清单：/src/cpu/x86/vm/bytecodeInterpreter_x86.hpp

功能：SET_LOCALS_SLOT 和 SET_STACK_SLOT 宏定义

```
// 入参 pc[1] 是 iload 指令后面的操作数，该操作数即为局部变量的 slot 索引号
// 该宏返回指定 slot 索引号的局部变量值
#define LOCALS_SLOT(offset)    ((intptr_t*)&locals[-(offset)])

// 将操作数栈栈顶指定位置的存储单元复制为 value
#define SET_STACK_SLOT(value, offset)    (*(intptr_t*)&topOfStack[-(offset)]
= *(intptr_t*)(value))
```

通过这 2 个宏定义可以看出，使用 C 语言所实现的 iload 字节码指令的逻辑，与模板解释器中直接使用本地机器指令所实现的逻辑是一致的，解释 iload 字节码指令时，都是先从 iload 指令中解析出跟随在该操作码之后的操作数，该操作数便是局部变量的 slot 索引号，解释器根据该索引号取出局部变量的值，最终将该数值传送到操作数栈栈顶。

由于字节码解释器使用 C/C++ 逻辑来解释字节码指令，使用 C/C++ 的解释逻辑在静态编译阶段所生成的机器指令，肯定要比模板解释器中人工生成的机器指令烦琐和冗长，因此解释效率相当低，所以如今的 JVM 不再使用它了，但是源码仍然保留。

本书将字节码解释器特地摘录出来说明一二，只是希望有兴趣的道友能够更加明白模板解释器的运作机制，说不定原来对模板解释器的概念不甚清楚，看了字节码解释器之后能够恍然大悟，则也是值得的。

9.4 栈顶缓存

前面讲过，当 JVM 使用模板解释器运行字节码指令时，最后生成的 iload_1 这条指令的本地机器码如下（32 位 x86 平台）：

```
mov    -0x4(%edi),%eax
```

HotSpot 提供了工具（HSDIS），可以使用该工具打印模板解释器为各个字节码指令所生成的本地机器码，可以据此验证实际生成的机器码是否与源码中所写的一致。HotSpot 在运行期所打印的 iload_1 的本地机器逻辑如下（32 位 x86 平台）：

```
iload_1 27 iload_1 [0xb36d8700, 0xb36d8740] 64 bytes
```

```
[Disassembling for mach='i386']
0xb36d8700: sub    $0x4,%esp
0xb36d8703: fstps  (%esp)
0xb36d8706: jmp    0xb36d8724
0xb36d870b: sub    $0x8,%esp
```

```

0xb36d870e: fstpl    (%esp)
0xb36d8711: jmp     0xb36d8724
0xb36d8716: push    %edx
0xb36d8717: push    %eax
0xb36d8718: jmp     0xb36d8724
0xb36d871d: push    %eax
0xb36d871e: jmp     0xb36d8724
0xb36d8723: push    %eax
0xb36d8724: mov     -0x4(%edi),%eax
0xb36d8727: movzbl  0x1(%esi),%ebx
0xb36d872b: inc     %esi
0xb36d872c: jmp     *-0x48f106a0(,%ebx,4)
0xb36d8733: nop
0xb36d8734: add     %al,(%eax)
0xb36d8736: add     %al,(%eax)
0xb36d8738: add     %al,(%eax)
0xb36d873a: add     %al,(%eax)
0xb36d873c: add     %al,(%eax)
0xb36d873e: add     %al,(%eax)

```

根据 `TemplateTable::iload(int n)` 这个 `iload_1` 字节码指令的生成器函数接口的代码逻辑，分析出只会生成 `mov -0x4(%edi),%eax` 这一条机器指令，可是 JVM 工具所打印出来的机器逻辑竟然有这么多，其中包含了 `mov -0x4(%edi),%eax` 这条指令，很奇怪吧！难道之前的分析存在问题？

为了进一步确认，可以再看一看 `iload` 这条字节码指令。前面讲过，模板解释器为该字节码指令所生成的本地机器逻辑如下（32 位 x86 平台）：

```

movzbl 0x1(%esi),%ebx
neg     %ebx
mov     (%edi,%ebx,4),%eax

```

再看看 JVM 工具在运行时所生成的指令，如下（32 位 x86 平台）：

```

iload 21 iload [0xb36d8420, 0xb36d8480] 96 bytes
[Disassembling for mach='i386']
0xb36d8420: sub     $0x4,%esp
0xb36d8423: fstps   (%esp)
0xb36d8426: jmp     0xb36d8444
0xb36d842b: sub     $0x8,%esp
0xb36d842e: fstpl   (%esp)
0xb36d8431: jmp     0xb36d8444
0xb36d8436: push    %edx
0xb36d8437: push    %eax
0xb36d8438: jmp     0xb36d8444
0xb36d843d: push    %eax
0xb36d843e: jmp     0xb36d8444
0xb36d8443: push    %eax

```

```

0xb36d8444: movzbl 0x1(%esi),%ebx
0xb36d8448: neg     %ebx
0xb36d844a: mov     (%edi,%ebx,4),%eax
0xb36d844d: movzbl 0x2(%esi),%ebx
0xb36d8451: add     $0x2,%esi
0xb36d8454: jmp     *-0x48f106a0(,%ebx,4)
0xb36d845b: mov     0x2(%esi),%ebx
0xb36d845e: bswap   %ebx
0xb36d8460: shr     $0x10,%ebx
0xb36d8463: neg     %ebx
0xb36d8465: mov     (%edi,%ebx,4),%eax
0xb36d8468: movzbl 0x4(%esi),%ebx
0xb36d846c: add     $0x4,%esi
0xb36d846f: jmp     *-0x48f106a0(,%ebx,4)
0xb36d8476: xchg    %ax,%ax
0xb36d8478: add     %al,(%eax)
0xb36d847a: add     %al,(%eax)
0xb36d847c: add     %al,(%eax)
0xb36d847e: add     %al,(%eax)

```

可以看到，`iload` 指令也存在同样的问题，实际所生成的本地机器逻辑与模板表中的生成器接口所设计的逻辑不一致。难道这里面真的存在什么问题不成？

事实上，模板表中所定义的生成器接口中所设计的逻辑没有错，JVM 工具打印出来的本地机器逻辑也没有错，所有的问题都与一个关键的优化措施有关，这个优化措施便是“栈顶缓存”。

可能绝大多数人第一眼看到栈顶缓存，脑子里都会打上一个大大的问号，这是个什么东西？栈顶就栈顶，为何还要加个缓存？缓存加在哪里？有什么好处？

在讲述 JVM 的栈顶缓存概念之前，先讲一讲 Java 开发中的缓存。有过 Java 开发及优化经验的道友基本都用过缓存技术，例如查询 DB 之前，通常会先查询缓存，如果缓存中没有，再去查询 DB。下面给出一段伪代码用于表述这种逻辑：

清单：/UserDao.java

功能：演示 Java 中查询 DB 时的缓存使用

```

class UserDao{
    public List<User> queryUserById(Long userId){
        List<User> userList = null;

        // 先查询缓存
        userList = cache.getByKey(userId);
        if(userList != null && userList.size() > 0){
            return userList;
        }
    }
}

```

```

// 如果在缓存中查询不到, 再查询 DB
userList = queryFromDB(userId);
if(userList != null && userList.size() > 0){

    // 如果从 DB 中查到数据, 则将数据保存到缓存
    cache.put(userId, userList);

    return userList;
}

return userList;
}
}

```

相信这一段示例程序对于大部分 Java 道友而言, 都不会陌生。在 Java 程序中使用缓存, 可以减轻 DB 负担, 从而提升应用程序的响应速度。这种缓存可以是本地缓存, 也可以是远程的缓存集群。但是不管是本地缓存还是远程的缓存中心, 数据应该都是存放在内存单元中, 肯定不可能存放在磁盘中, 否则还不如直接查询 DB。JVM 中的所谓栈顶缓存, 与之完全不同。栈顶缓存的数据通过寄存器来暂存, 并非内存。要明白这一点, 还得懂一点计算机的硬件知识。对于 CPU 而言, 一方面, 就读取速度而言, CPU 从寄存器中读取速度最快, 其次是内存, 再次是磁盘。CPU 从寄存器中读取数据的速度往往比从内存中读取要快上好几个数量级 (例如百倍), 这种速度方面的差异非常大, 毕竟相差百倍以上, 这是任何一个优秀的系统设计师都不能避免的问题。另一方面, CPU 在执行运算时, 例如做加法运算, 并不能直接对两个内存中的数据直接进行求和, 要么将两个数据全部从内存读取到寄存器中然后对两个寄存器进行求和, 要么将一个数据读取进寄存器而另一个保留在内存中, 这种规则本身没有问题, 问题的关键在于, 寄存器的数量是相当稀少的, 一个 CPU 能够集成几十个寄存器便已经是奢侈至极, 比起现代计算机的内存动辄就是 8GB、16GB 甚至 32GB 的巨大空间, 这真的是沧海一粟。由于寄存器的数量很稀少, 因此 CPU 往往在空间和效率上不能两全。例如要对两个数据进行求和, 最快的方式当然是 CPU 直接对内存中的两个数据累加, 但是 CPU 并不支持这种运算, 因此在对两个数据进行求和之前, 只能将其中一个数据先读取进寄存器, 或者将两个数据全部读进寄存器然后才能进行累加。这中间必然存在效率上的牺牲。既然对内存数据进行运算这么慢, 那将数据全部放进寄存器不就行了吗? 这种方式并不可行, 因为刚刚讲过, 寄存器的数量是极其稀少的, 因此编译器在将高级语言编译为机器语言时, 会将数据加载进内存, 而非全都加载进寄存器。

JVM 的栈顶缓存正是针对 CPU 这种在时间与空间上不能两全的遗憾而进行的改进措施, 当然, 这种改进措施蕴含了计算机运行机制的精华, 也是 JVM 执行引擎的精华。

由于 CPU 无法同时兼顾时间与空间, 而 JVM 追求的则是性能, 因此只能舍弃空间, 这种取舍的结果便是, 模板解释器在执行操作数栈操作时, 如果按照常规的思路, 肯定会将数据直

接压入栈顶, 栈顶其实也是内存存储单元, 但是 JVM 并没有走寻常路, 为了追求性能, JVM 在执行操作数栈相关操作时, 会优先将数据传送到寄存器, 而非真正的栈顶。在后续流程中, CPU 执行运算时, 便无须将数据再从栈顶传送到寄存器, 因为数据本来就缓存在寄存器中, 这便节省了一次内存读写, 从而提升了 JVM 虚拟机运算指令的执行效率。这便是栈顶缓存。这种机制与 java 应用开发中使用缓存中间件的思路类似, 只不过对于栈顶缓存而言, 缓存介质是硬件寄存器, 而非内存单元。

由于寄存器的数量十分有限, 多的也就几十个, 因此 JVM 并不是每次都能将数据存进寄存器, 所以在将数据存进寄存器 (即栈顶缓存) 之前, 必须先判断寄存器中是否有数据, 如果有数据, 得先想办法将数据移走, 然后才能将当前操作的数据存进去。另外, Java 内建的数据类型很丰富, 因此栈顶缓存的数据类型也是五花八门, 什么都有, JVM 在将这些数据移走时必须考虑真实的数据类型, 对待不同的数据类型处理逻辑也是不同的, 所以 JVM 在解释一个字节码指令时, 需要包含处理栈顶不同类型数据的逻辑, 这便是前文使用 JVM 工具打印运行时所输出的 `iload_1` 和 `iload` 这 2 个字节码指令的本地机器码时代码比想象中要多得多的原因。这一点其实与 Java 应用开发的缓存策略类似, 在 Java 应用开发中, 如果要查询满足某种条件的数据, 则需要考虑缓存中是否有数据, 如果有, 就从读取缓存的入口进入获取结果集, 否则就从读取 DB 的入口进入获取结果集。很多应用为了进一步提升性能, 会考虑两级缓存——本地缓存和远程缓存, 本地缓存通常缓存热点数据, 将热点数据加载到本地内存, 从而避免访问远程缓存中心时的网络开销, 从而在命中率超过 80% 的情况下, 应用的整体性能将会提升。因此, 在 Java 应用开发中要访问数据, 通常也会有很多“入口点”, 如果本地一级缓存中有数据, 就从中取数; 否则就判断远程缓存中是否有, 若有, 则从远程缓存中取数; 否则就只能查询 DB。这些查询一级缓存和二级缓存的前置逻辑, 便类似于 Java 字节码指令处理的前置逻辑, 都是处理缓存。

以 `iload_1` 这条字节码指令为例, 当栈顶缓存为空时, 则 JVM 会直接将 slot 索引号为 1 的局部变量传送到寄存器 `ax` 中 (x86 平台), 并不需要先将栈顶数据移出去, 因此 JVM 在运行期会直接从如下入口进入:

```
iload_1 27 iload_1 [0xb36d8700, 0xb36d8740] 64 bytes
```

```
[Disassembling for mach='i386']
```

```
0xb36d8700: sub    $0x4,%esp
0xb36d8703: fstps  (%esp)
0xb36d8706: jmp    0xb36d8724
0xb36d870b: sub    $0x8,%esp
0xb36d870e: fstpl  (%esp)
0xb36d8711: jmp    0xb36d8724
0xb36d8716: push   %edx
0xb36d8717: push   %eax
0xb36d8718: jmp    0xb36d8724
```



```

0xb36d871d: push  %eax
0xb36d871e: jmp   0xb36d8724
0xb36d8723: push  %eax
0xb36d8724: mov   -0x4(%edi),%eax          =====>直接从这一句开始解释
0xb36d8727: movzbl 0x1(%esi),%ebx
0xb36d872b: inc   %esi
0xb36d872c: jmp   *-0x48f106a0(,%ebx,4)
0xb36d8733: nop
0xb36d8734: add   %al, (%eax)
0xb36d8736: add   %al, (%eax)
0xb36d8738: add   %al, (%eax)
0xb36d873a: add   %al, (%eax)
0xb36d873c: add   %al, (%eax)
0xb36d873e: add   %al, (%eax)

```

而当栈顶缓存不为空时，例如栈顶缓存此时已经存储了一个 int 型的数据，则 JVM 在执行 `iload_1` 字节码指令时，需要先将栈顶缓存中的数据移到其本来应该存储的内存位置，然后再将 slot 索引号为 1 的局部变量传送到栈顶缓存（即寄存器）中，此时，`iload_1` 的进入点如下：

```
iload_1 27 iload_1 [0xb36d8700, 0xb36d8740] 64 bytes
```

```

[Disassembling for mach='i386']
0xb36d8700: sub    $0x4,%esp
0xb36d8703: fstps  (%esp)
0xb36d8706: jmp    0xb36d8724
0xb36d870b: sub    $0x8,%esp
0xb36d870e: fstpl  (%esp)
0xb36d8711: jmp    0xb36d8724
0xb36d8716: push   %edx
0xb36d8717: push   %eax
0xb36d8718: jmp    0xb36d8724
0xb36d871d: push   %eax
0xb36d871e: jmp    0xb36d8724
0xb36d8723: push   %eax          =====>直接从这一句开始解释
0xb36d8724: mov    -0x4(%edi),%eax
0xb36d8727: movzbl 0x1(%esi),%ebx
0xb36d872b: inc    %esi
0xb36d872c: jmp    *-0x48f106a0(,%ebx,4)
0xb36d8733: nop
0xb36d8734: add    %al, (%eax)
0xb36d8736: add    %al, (%eax)
0xb36d8738: add    %al, (%eax)
0xb36d873a: add    %al, (%eax)
0xb36d873c: add    %al, (%eax)
0xb36d873e: add    %al, (%eax)

```

这一次的入口点变成了 `push %eax`，其作用是将 `eax` 寄存器（即栈顶缓存）中的数据先推送

至操作数栈栈顶，然后再接着执行 `iload_1` 本身的逻辑。

那么 `iload_1` 在什么情况下，在执行之前，栈顶缓存是空的呢？可以看下面这个例子：

清单：/Test.java

功能：示例 `iload_1` 执行之前栈顶缓存为空的情况

```
class Test{
    public static int add(int x, int y){
        int z = 8;
        int sum = y + z;
        return sum;
    }
}
```

编译该 Java 类并使用 `javap` 命令分析编译后的字节码文件，输出如下：

```
public static int add(int, int);
descriptor: (II)I
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=4, args_size=2
       0: bipush      8
       2: istore_2
       3: iload_1
       4: iload_2
       5: iadd
       6: istore_3
       7: iload_3
       8: ireturn
```

观察这一段字节码指令，在执行 `iload_1` 之前，先执行了 `bipush 8` 和 `istore_2`，这 2 条字节码指令将完成为变量 `z` 赋值的逻辑。`istore_2` 字节码指令执行完成之后，由于栈顶并没有数据等待操作，用不着将数据缓存在寄存器中，因此此时寄存器中没有数据。在这种情况下，当执行 `iload_1` 指令时，JVM 便会直接从 `iload_1` 的本地机器码 `mov -0x4(%edi),%eax` 开始执行。

修改上面的 `Test` 类，改成如下：

清单：/Test.java

功能：示例 `iload_1` 执行之前栈顶缓存不为空的情况

```
class Test{
    public static int add(int x, int y){
        int z = 8;
        int sum = x + y;
        return sum;
    }
}
```

编译该 Java 类并使用 `javap` 命令分析编译后的字节码文件，输出如下：

```
public static int add(int, int);
  descriptor: (II)I
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=4, args_size=2
       0: bipush      8
       2: istore_2
       3: iload_0
       4: iload_1
       5: iadd
       6: istore_3
       7: iload_3
       8: ireturn
```

注意观察，此时在执行 `iload_1` 指令之前，JVM 必须先执行 `iload_0` 这条指令。`iload_0` 指令执行完之后，由于使用了栈顶缓存策略，因此 JVM 会将 slot 索引号为 0 的局部变量直接传送到寄存器指针（即栈顶缓存）中，而不是直接将该局部变量推送至栈顶，如此一来，栈顶缓存（即寄存器）中便有数据了，并且是 `int` 类型的。接着 JVM 开始执行 `iload_1` 这条指令，由于栈顶缓存中已经有数据，因此 JVM 必须先将 `iload_0` 所加载的数据从栈顶缓存中移走，所以 JVM 执行 `iload_1` 指令时就必须从 `push %eax` 这条机器码开始执行，该指令会将 `iload_0` 指令加载到栈顶缓存的数据推送至操作数栈栈顶。

以上便是栈顶缓存的基本原理。其实栈顶缓存远不止这么简单，但是要想深入研究该技术，必须对机器指令及计算机底层非常熟悉，笔者不知道各位道友对此是否足够感兴趣，因此关于栈顶缓存的话题先讲这么多。如果大家果真求知欲特别强烈，笔者会探本求原，深入揭示栈顶缓存的来龙去脉。

9.5 栈式指令集

Java 的字节码指令都是面向栈的，面向栈的指令集往往有一个特点：不需要指定操作数，专业术语就是“零地址”指令。例如，在 Java 中对两个 `int` 类型的数据求和，其对应的字节码指令是 `iadd`。乍一看，这种指令很是让人感觉“无厘头”，不是说好的对两个数求和的吗，数呢？这种指令的写法与通常意义上我们所见到的加法表达式格式相去甚远，因此有点让人莫名其妙。在数学中，对两个数求和，司空见惯的一种表达式肯定是下面这种：

$$x = y + z$$

这种表达式非常简易明了，随便一个人一看都知道是在对两个数求和。不仅计算器讲究封

装的艺术，数学也讲究封装，而事实上，数学里的封装要算得上计算机封装的老祖宗了，现代计算机的诞生其实是数学概念抽象的结果，抽象便是封装。将上面这个数学表达式抽象成一个数学函数，变成：

```
add(x, y, z)
```

计算机本来就是用于处理数字信号的，对于这种函数，某些 CPU 天生能支持，例如，ARM 处理器。CPU 在处理这种数学函数时，会通过约定的指令格式来完成。例如，上面这个函数翻译成计算器指令，可以如下：

```
add x, y, z
```

这便是物理计算机支持的硬件指令，这种指令能够对操作码后面的两个数求和，并将累加结果保存到操作码后面的第一个操作数中。将该指令的表达形式进行抽象，可以得到下面这种一般意义上的指令格式：

```
op dest, src1, src2
```

这种指令格式的含义是：对 `op` 操作码后面的两个操作数 `src1` 和 `src2` 进行某种操作，并将操作结果保存到 `op` 操作码后面的第一个操作数 `dest` 中。有了这种通用的表达式，计算机便能支持各种数学运算，例如，对两个数执行减法运算，指令如下：

```
sub dest, src1, src2
```

其实，类似“`op dest, src1, src2`”这种格式的指令，在计算机中叫作“三地址”指令，所谓三地址指令，其实就是指操作码 `op` 的后面跟了 3 个操作数。由于在计算机内部，数据都存储在内存或寄存器中，因此 `op` 操作码后面的操作数通常都是指某个内存单元编号或者某个特定的寄存器，所以叫作“三地址”。

不过有些 CPU 设计者认为三地址指令太长，需要简化。例如， $x = y + z$ 这种表达式可以简化成 $y += z$ ，这种表达式仍然能够对两个数进行求和，并将求和结果保存到其中一个数字之中。对 $y += z$ 表达式重新编排，可以写成下面这种格式：

```
+= y, z
```

乍一看，这种格式很奇怪，但是如果将其使用数学函数来表达，则可以抽象成下面这种函数：

```
add(y, z)
```

这样的数学函数表达式大家都懂。如果让计算机指令来支持这种数学函数，则可以写成下面这种格式：

```
add y, z
```

x86 系列的处理器便能够直接执行这种指令对两个数进行求和。若使用 C 语言编写程序计算 $x = y + z$, 则在 x86 平台上编译后便会生成类似上面这条指令的机器码。将该指令进行抽象, 可以得到下面这种通用的指令格式:

```
op dest, src
```

这种指令的含义是: 对 op 操作码后面的两个操作数进行某种运算, 并将运算结果保存在 dest 第一个操作数中。这种格式的指令使用专业术语描述叫作“二地址”指令, 意思很明确, 操作码后面跟了 2 个操作数。相比于三地址指令, 二地址指令所需要的内存空间变小了, 并且整个指令也更加精简。

上面这两种指令格式, 无论是三地址格式还是二地址格式, 就指令格式本身而言, 普罗大众都比较容易根据指令格式而明白其功能, 并且这种格式也符合数学领域中的函数抽象, 相比于 Java 中的 iadd 这种“零地址”的指令格式, 更加容易被人接受。其实, “零地址”这种格式的指令会让人产生慌乱, 慌乱的原因是: 不知道这种指令对谁执行累加。而无论是二元地址还是三元地址的指令, 至少所操作的目标数据都包含在指令中。

事实上, 大凡设计成“零地址”格式的指令集, 通常都是面向栈的指令集, 既然是面向栈的, 则指令所操作的源数据和目标数据, 便默认存放于栈上。而上面二元地址和三元地址的指令集, 更多的是基于寄存器的架构, 所操作的数据直接位于寄存器中 (当然也有部分位于内存单元中)。之所以二元和三元地址指令集大多数直接面向寄存器, 是因为这种多元地址指令后面所跟随的操作数可以直接被指定为目标寄存器, 而 CPU 读取寄存器的性能要远远高于内存, 因此能够直接基于寄存器运算的指令, 没必要设计成面向栈式操作, 否则反而不美。而零地址指令往往不能设计成面向寄存器操作, 或者说设计难度很大, 因为不同的硬件平台, 其所集成的寄存器数量、内部标识、指令格式并不相同, 而零地址指令由于并不直接包含操作数, 因此无法明确指定所操作的数据到底位于哪个寄存器中, 所以不能很容易地设计成面向寄存器操作, 只能设计成基于栈操作, 因为不管何种计算器, 栈的概念都是被支持的, 并且语义也是一致的。同时由于这种特性, 因此寄存器式指令集一般是硬件平台相关的, 而栈式指令集则能跨平台。例如, 在 x86 平台上执行两个数据相加, 可以直接在指令中指定所操作的元数据位于哪个寄存器, 以及结果数据位于哪个寄存器, 例如下面这条指令:

```
add %ax, %bx
```

这条求和指令直接指定源数据分别位于 ax 和 bx 这两个通用寄存器中, 并且操作结果存储在 bx 寄存器中 (AT&T 语法)。而如果使用栈式指令进行求和, 例如 Java 中的 iadd 指令, 假设该指令是基于寄存器的, 那么问题来了, iadd 本身就是一条完整的指令, 并不包含任何操作数, 那么其所操作的源数据到底在哪两个寄存器中呢? ax? bx? 不知道, 32 位的 x86 CPU 有 8 个 32 位通用寄存器, 而 SUN 的 SPARC 处理器则有 24 个通用寄存器。寄存器是如此之多, 而 iadd

指令又无法指定所操作的源数据位于哪里,因此如果硬生生将零地址指令设计成面向寄存器的,难度可想而知。

进一步说,即使规定默认的寄存器地址而将基于栈的指令集设计成面向寄存器的架构,其灵活性也会大打折扣。例如,在 x86 平台上能够支持直接对一个内存存储单元和一个寄存器求和,例如:

```
add (%ax), %bx
```

该指令将 ax 寄存器所指的内存地址的数据与 bx 寄存器进行累加,并将累加结果保存进 bx 寄存器中(AT&T 语法)。对于这种灵活的求和指令,很显然,零地址的指令集根本支持不了。JVM 由于刚“出道”时便以“跨平台”作为最大的特性大力宣传,因此其指令集便选择了零地址的格式,而零地址的指令集又只能选择基于栈的架构,实在是有其深刻的技术制约因素。由于指令集是面向栈的,因此 JVM 的执行引擎内部也只能紧紧围绕栈来实现,上文所讲的栈顶缓存便是针对堆栈而进行的一项优化措施,其优化的思路还是优先使用寄存器。需要注意的是,这里所言的栈是指“求值栈”,而不是指系统调用栈(system call stack)。有些虚拟机把求值栈实现在系统调用栈上,但两者概念上不是一个东西。

栈式指令集既然是零地址格式,那就意味着其操作的源数据与目的数据皆位于求值栈栈顶。既然所操作的数据位于栈顶,那么在操作之前,必然会有指令将数据先传送到栈顶,所以使用一条寄存器式指令便能实现的逻辑,往往需要多条栈式指令方能实现。例如,在 JVM 执行 iadd 指令进行求和之前,必定会有 iload 系列或者 iconst 或者 bipush 等将数据推送至栈顶的前置指令。例如下面这个示例:

清单: /Test.java

功能: 演示 iadd 指令

```
class Test{
    public static void main(String[] args) {
        int a = 66;
        int b = 87;
        int sum = a + b;
    }
}
```

编译该类并使用 javap 命令分析编译后的字节码文件,分析结果输出如下:

```
public static void main(java.lang.String[]);
  descriptor: ([Ljava/lang/String;)V
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=2, locals=4, args_size=1
       0: bipush      66
```



```

2: istore_1
3: bipush      87
5: istore_2
6: iload_1
7: iload_2
8: iadd
9: istore_3
10: return

```

观察这段字节码指令，在 `iadd` 指令之前，存在两条前置指令 `iload_1` 和 `iload_2`，这两条指令分别将 `slot` 索引号为 1 和 2 的 `int` 类型的数据推送至操作数栈栈顶，这两个推送的数据其实便是接下来 `iadd` 指令的源操作数。那么 `iadd` 指令执行完成之后，所求取的结果数据存储在哪里呢？这得先从字节码指令执行的原理说起。

前文讲过，JVM 在启动阶段会为所有字节码指令生成本地机器码指令。而本地机器码指令大多数都是基于寄存器的。从这个角度来看，JVM 的栈式指令集其实只能算是“伪指令”，因为一方面，JVM 并不具备真正具有运算能力的硬件数字电路，另一方面，JVM 的字节码指令最终仍然需要基于寄存器架构的本地机器指令才能执行。对于 `iadd` 指令而言，使用 `HSDIS` 工具在运行期打印该指令的本地指令如下（32 位 x86 平台）：

```
iadd 96 iadd [0xb36d9f40, 0xb36d9f60] 32 bytes
```

```
[Disassembling for mach='i386']
```

```

0xb36d9f40: pop    %eax
0xb36d9f41: pop    %edx
0xb36d9f42: add    %edx,%eax
0xb36d9f44: movzbl 0x1(%esi),%ebx
0xb36d9f48: inc    %esi
0xb36d9f49: jmp    *-0x48f106a0(,%ebx,4)

```

如果你熟悉汇编，上面这段代码一看便能明白其逻辑：先执行 `pop %eax` 和 `pop %edx` 指令将栈顶的两个数据分别弹出至 `eax` 和 `edx` 这两个寄存器中，接着执行 `add %edx,%eax` 指令对这两个数据进行求和，并将求和结果再存储进 `eax` 寄存器中（AT&T 语法）。这样便完成了累加的运算。不过运算完成之后，结果被存储在了 `eax` 寄存器中，其实这里仍然使用了栈顶缓存策略——优先将数据存储进寄存器。但是事实上，按照 JVM 内部对 `iadd` 指令的定义，该指令执行完成之后，数据应该会被存储进栈顶——因为整个 JVM 的执行引擎和指令集都是面向栈操作的。

前面讲了 JVM 的指令集为何是面向栈的，以及为何被设计成零地址格式和栈式指令集的实现方式。总体而言，虚拟机选择栈式指令集有利有弊，好处大抵有如下几个：

- ◎ 编译器更加容易设计和实现。因为不用考虑一堆寄存器了，只需要关注栈顶即可。
- ◎ 指令集非常精简小巧。因为大部分指令都是零地址格式，一个字节就能完成某种

操作，所占内存空间也小。

- ◎ 跨平台。虽然不同的硬件平台，其寄存器和机器级别的指令格式不同，但是 JVM 不管这些，只关注栈，栈总是所有硬件平台都要支持的一种内存结构，或者说即使没有栈，只要有内存即可。

但是得到这些好处的同时，也带来了如下硬伤：

- ◎ 实现同样的功能，需要更多的指令。以求和为例，基于寄存器的指令集只需执行 `add dest, src` 这样的指令便能完成求和，而栈式指令集则需要先 `load` 再 `add`，需要多条指令。所以虽然单个指令变小了，但是完成一个功能需要更多的指令，总体上其实并没有精简太多，甚至局部反而更加烦琐。
- ◎ 性能下降。这是由于栈式指令集需要更多的指令才能完成某个功能，并且完成同一个功能所对应的本地机器码比原本使用纯粹的基于寄存器的机器指令更多，CPU 需要更多的时钟周期，需要制造更多电子脉冲。

前面说过，JVM 之所以使用栈式指令集，是为了跨平台，但是同样使用 Java 语法规则的 Android 技术体系，却直接使用了寄存器式指令集。安卓 4 之前，安卓使用 Dalvik 虚拟机来解释执行安卓字节码文件，虽然 HotSpot 与 Dalvik 同样都是虚拟机，但是为何 Dalvik 没有使用栈式指令集呢？问题就出在跨平台上。安卓刚“出道”时，所针对的就是 ARM 系列处理器，该处理器有 16 个 32 位通用寄存器，而 Dalvik 里面则设计了 16 个虚拟寄存器，在运行期，Dalvik 会将虚拟寄存器全部映射到物理寄存器，从而让 Dalvik 能够在 ARM 处理器上高效执行，从而充分发挥寄存器架构的优势。所以安卓系统一开始仅仅是借用了 Java 跨平台的语法，但是并没有为其打造一颗跨平台的“心脏”，所以安卓并不能直接移植到其他异构的手机平台上。

由于 JVM 的性能比较低，因此大神们想尽了各种办法来优化性能，各种奇思妙想层出不穷，各路大招层见迭出，所用技能让人叹为观止！HotSpot 内部解释器从最原始的字节码解释器（以 C 语言函数逻辑解释执行字节码）升级到模板解释器（直接生成本地机器码），解决了原始解释器效率低下的突出问题（很严重，严重到被 C/C++ 程序员嘲笑）。接着加入了 JIT 编译器，其能够在运行期针对热点代码进行即时编译，JIT 使用了多种优化策略使得编译出来的代码指令更高效，例如将代码内联减少字节码跳转次数，能够加快热点代码的运行效率。HotSpot 还针对客户端和服务端分别开发了 C1 和 C2 两层编译优化功能，C1 和 C2 属于动态自适应编译器，其中 C1 编译器仅做了简单优化，这主要是因为客户端的 JVM 虚拟机追求快速启动、快速响应，如果编译时间过长，反而影响客户端体验，而 C2 则会做深层次的优化，这种编译器所编译出来的代码质量较高，但是因此需要较长的编译时间成本，所以仅适合用于服务端 JVM。JIT、C1、C2 这些优化技术太过于底层，并且需要很深厚的编译原理内功，有兴趣的道友可以一起

研究。

但是对于 JIT 等优化技术也不能过于迷信,使用 JIT 编译出来的代码指令的执行效率并不一定就比直接解释执行的效率高,甚至如果一个方法并不是经常被调用,那么可能执行一次 JIT 编译的时间成本都要高于直接解释执行一次该方法的时间。但是很多时候使用 JIT 编译所获得的代码质量,却比使用 C/C++代码完成同样的 Java 逻辑的代码质量还要高,这才是 JIT 迷人的地方。

这里举一个例子来比较 JVM 使用模板解释器和使用 JIT、C1 和 C2 编译器所生成的本地代码,Java 示例如下:

清单: /Test.java

功能: 比较 JVM 使用模板解释器和使用 JIT、C1 和 C2 编译器所生成的本地代码

```
public class Test{
    public static int calc(int i, int j){
        int b = 0x211;
        int sum = i + b;
        sum = sum - j;
        b = sum * i;
        sum = j * (b - 5);
        return sum;
    }

    public static void main(String[] args)throws Exception{
        for(int k = 0; k < 100000; k++){
            calc(k, k+1);
        }
    }
}
```

该示例的 main()主函数中使用 for 循环重复调用 calc(int, int)方法,并且循环了 10 万次。所以要循环这么多次,是为了让 JVM 能够“侦查”到 calc(int, int)方法是一个热点代码。对于热点代码,HotSpot 会调用 JIT 即时编译器将其编译成高质量的本地代码。同时,在 calc(int, int)方法里面进行了比较复杂的运算,之所以设计得比较复杂,是为了接下来要比较 C1 和 C2 这两款编译器所生成的本地机器指令的质量,如果算法简单,JVM 可能压根就不开启 C2 分层优化编译器,这样便无法比较。

使用 HSDIS 插件可以查看运行后 JIT 所生成的本地汇编指令,所生成的本地汇编代码分为 C1 版和 C2 版(即 C1 编译器和 C2 编译器分别生成的本地指令),其中 C1 版如下:

```
CompilerOracle: print *Test.calc
Java HotSpot(TM) 64-Bit Server VM warning: printing of assembly code is
enabled; turning on DebugNonSafepoints to gain additional output
```

```

Compiled method (c1)      103   35       3      Test::calc (24 bytes)
total in heap [0x000000010eaa3b90,0x000000010eaa3ea0] = 784
relocation    [0x000000010eaa3cb8,0x000000010eaa3ce0] = 40
//.....
Loaded disassembler from HSDIS-amd64.dylib
Decoding compiled method 0x000000010eaa3b90:
Code:
[Disassembling for mach='i386:x86-64']
[Entry Point]
[Verified Entry Point]
[Constants]
# {method} {0x00000001276f0330} 'calc' '(II)I' in 'Test'
# parm0:   rsi      = int
# parm1:   rdx      = int
#          [sp+0x40] (sp of caller)
0x000000010eaa3ce0: mov    %eax,-0x14000(%rsp)
0x000000010eaa3ce7: push   %rbp
0x000000010eaa3ce8: sub    $0x30,%rsp
0x000000010eaa3cec: movabs $0x1276f04e0,%rax ; {metadata(method data
for {method} {0x00000001276f0330} 'calc' '(II)I' in 'Test')}
0x000000010eaa3cf6: mov    0xdc(%rax),%edi
0x000000010eaa3cfc: add    $0x8,%edi
0x000000010eaa3cff: mov    %edi,0xdc(%rax)
0x000000010eaa3d05: movabs $0x1276f0330,%rax ; {metadata({method}
{0x00000001276f0330} 'calc' '(II)I' in 'Test'))}
0x000000010eaa3d0f: and    $0x1fff8,%edi
0x000000010eaa3d15: cmp    $0x0,%edi
0x000000010eaa3d18: je     0x000000010eaa3d3e ;*sipush
                                ; - Test::calc@0 (line 35)

0x000000010eaa3d1e: mov    %rsi,%rax
0x000000010eaa3d21: add    $0x211,%eax
0x000000010eaa3d27: sub    %edx,%eax
0x000000010eaa3d29: imul   %esi,%eax
0x000000010eaa3d2c: sub    $0x5,%eax
0x000000010eaa3d2f: imul   %edx,%eax
0x000000010eaa3d32: add    $0x30,%rsp
0x000000010eaa3d36: pop    %rbp
0x000000010eaa3d37: test   %eax,-0x20c0c3d(%rip) # 0x000000010c9e3100
                                ; {poll_return}

0x000000010eaa3d3d: retq
0x000000010eaa3d3e: mov    %rax,0x8(%rsp)
0x000000010eaa3d43: movq   $0xffffffffffffffff, (%rsp)
0x000000010eaa3d4b: callq  0x000000010ea875e0 ; OopMap{off=112}
                                ;*synchronization entry
                                ; - Test::calc@-1 (line 35)
                                ; {runtime_call}

```

```

0x0000000010eaa3d50: jmp     0x0000000010eaa3d1e
0x0000000010eaa3d52: nop
0x0000000010eaa3d53: nop
0x0000000010eaa3d54: mov     0x2a8(%r15),%rax
0x0000000010eaa3d5b: movabs  $0x0,%r10
0x0000000010eaa3d65: mov     %r10,0x2a8(%r15)
0x0000000010eaa3d6c: movabs  $0x0,%r10
0x0000000010eaa3d76: mov     %r10,0x2b0(%r15)
0x0000000010eaa3d7d: add     $0x30,%rsp
0x0000000010eaa3d81: pop     %rbp
0x0000000010eaa3d82: jmpq    0x0000000010e9f5b20 ; {runtime_call}
0x0000000010eaa3d87: hlt
// ...
0x0000000010eaa3d9f: hlt
[ExceptionHandler]
// ...

```

观察上面 C1 编译器所生成的机器指令，HSDIS 工具给出 Test.calc()方法入参的载体，这两个入参的载体分别是：

```

# parm0:    rsi        = int
# parm1:    rdx        = int

```

这表示 Test.calc(int i, int j)的入参 i 的值将被从 main()主函数传递给 rsi 寄存器，而入参 j 的值将被传递给 rdx 寄存器。C1 编译器所生成的核心算法指令从地址 0x0000000010eaa3d1e 开始，到地址 0x0000000010eaa3d2f 结束，一共包含 6 条机器指令，如下：

```

0x0000000010eaa3d1e: mov     %rsi,%rax
0x0000000010eaa3d21: add     $0x211,%eax
0x0000000010eaa3d27: sub     %edx,%eax
0x0000000010eaa3d29: imul    %esi,%eax
0x0000000010eaa3d2c: sub     $0x5,%eax
0x0000000010eaa3d2f: imul    %edx,%eax

```

这 6 条机器指令完成 Test.calc(int, int)方法内部的算法逻辑，懂汇编的道友一看就会明白这 6 条指令的含义，的确与 Test.calc(int, int)方法的逻辑完全保持一致。这 6 条指令之前有 11 条机器指令完成相关的准备工作，这 6 条指令执行完成之后，再执行 3 条指令便会执行 retq 指令而结束。

再看看 C2 版的汇编指令：

```

Compiled method (c2)      105   36      4      Test::calc (24 bytes)
total in heap [0x0000000010eaa6b10,0x0000000010eaa6d00] = 496
relocation    [0x0000000010eaa6c38,0x0000000010eaa6c40] = 8
// ...
dependencies  [0x0000000010eaa6cf8,0x0000000010eaa6d00] = 8
Decoding compiled method 0x0000000010eaa6b10:

```

```

Code:
[Entry Point]
[Verified Entry Point]
[Constants]
# {method} {0x00000001276f0330} 'calc' '(II)I' in 'Test'
# parm0:   rsi      = int
# parm1:   rdx      = int
#          [sp+0x20] (sp of caller)
0x000000010eaa6c40: sub    $0x18,%rsp
0x000000010eaa6c47: mov    %rbp,0x10(%rsp)    ; *synchronization entry
                                ; - Test::calc@-1 (line 35)

0x000000010eaa6c4c: mov    %esi,%eax
0x000000010eaa6c4e: sub    %edx,%eax
0x000000010eaa6c50: add    $0x211,%eax
0x000000010eaa6c56: imul   %esi,%eax
0x000000010eaa6c59: add    $0xfffffffffffffffb,%eax
0x000000010eaa6c5c: imul   %edx,%eax            ; *imul
                                ; - Test::calc@20 (line 39)

0x000000010eaa6c5f: add    $0x10,%rsp
0x000000010eaa6c63: pop    %rbp
0x000000010eaa6c64: test   %eax,-0x20c3c6a(%rip)    # 0x000000010c9e3000
                                ; {poll_return}

0x000000010eaa6c6a: retq
0x000000010eaa6c6b: hlt
// ...
0x000000010eaa6c7f: hlt
[ExceptionHandler]
[Stub Code]
// ...

```

C2 版的指令数量总体上比 C1 版的要少很多，但是主要的算法逻辑并没有精简，依然由 6 条机器码指令完成，如下：

```

0x000000010eaa6c4c: mov    %esi,%eax
0x000000010eaa6c4e: sub    %edx,%eax
0x000000010eaa6c50: add    $0x211,%eax
0x000000010eaa6c56: imul   %esi,%eax
0x000000010eaa6c59: add    $0xfffffffffffffffb,%eax
0x000000010eaa6c5c: imul   %edx,%eax

```

这 6 条指令也用于完成 Test.calc(int, int) 方法的算法逻辑，但是这 6 条指令与上面 C1 版的 6 条指令有所不同，调整了部分顺序，并且使用了补码操作代替减法运算，可以算作部分优化。但是由于 C2 版的机器指令数量远远少于 C1 版的机器指令数量，因此 C2 编译器的编译质量更高。但是无论是 C1 编译器还是 C2 编译器，其编译后的本地机器码整体质量大多（不能绝对）

比直接使用 JVM 内置的模板解释器所生成的本地机器码的质量要高很多，而影响 JVM 内置的模板解释器效率的一个核心因素便是字节码指令的分发，字节码指令的分发需要对应的 `jmp` 机器指令才能完成，有几条字节码指令就会有几次 `jmp`，这种跳转一方面使得模板解释器生成的本地机器码数量增多，另一方面也降低了执行效率。

使用 `javap` 命令输出 `Test.calc(int, int)` 方法所对应的字节码指令如下：

```
public static int tt(int, int);
  descriptor: (II)I
  flags: ACC_PUBLIC, ACC_STATIC
  Code:
    stack=3, locals=4, args_size=2
       0: sipush      529
       3: istore_2
       4: iload_0
       5: iload_2
       6: iadd
       7: istore_3
       8: iload_3
       9: iload_1
      10: isub
      11: istore_3
      12: iload_3
      13: iload_0
      14: imul
      15: istore_2
      16: iload_1
      17: iload_2
      18: iconst_5
      19: isub
      20: imul
      21: istore_3
      22: iload_3
      23: ireturn
```

通过输出结果可知，`Test.calc(int, int)` 方法一共对应 24 条字节码指令，假设每条字节码指令仅对应 4 条本地机器码（4 条机器码指令是最少的了，仅仅 `dispath_next` 分发就需要占用 3 条机器码），那么 24 条字节码指令至少也会生成 96 条本地机器码，这种级别的数量相比于上面 C1 和 C2 编译器所生成的本地机器码数量而言，无疑是巨大的。由此可见，C1 和 C2 这种分层自适应编译器所带来的性能提升当真不是说着玩的，而是真刀真枪的，能够将性能提升几个数量级。

除了这些优化技术，HotSpot 还使用一些内存分配、并发控制方面的优化技术，其中比较突出的是“逃逸分析”。所谓“逃逸分析”是指当一个 Java 对象被定义后，可能会被外部方法引用，例如被当作参数传递到其他方法中，这称为“方法逃逸”；也可能被其他线程访问，这个称

为“线程逃逸”。若能证明一个 Java 不会逃逸到其他方法中，则在为该对象分配内存空间时，可以直接进行“栈上分配”，即直接将 Java 对象实例分配在栈中，而非堆内存，这种分配策略所带来的一个直接好处便是不需要通过 GC 来回收对象实例，当 Java 方法调用完成，方法栈被回收时，Java 对象实例也跟着被销毁。其实这种分配方式在 C/C++ 中都有对应的实现，在 C 语言中，如果想在栈上直接分配一个结构体（姑且将 C 语言中的结构体看作一个对象，其实无论结构体还是类型，本质上都可以认为是一种复合的数据结构），可以直接通过 `struct A a`（假设 A 是一种自定义的结构体）这样的方式来声明；而如果想在堆中分配结构体实例，则需要通过 `malloc()` 函数来实现。在 C++ 中要实现栈上分配和堆上分配就更简单了，就看你创建对象实例时是否使用 `new` 关键字了。在 HotSpot 中实现了这种“栈上分配”技术，当确定一个 Java 类不会逃逸到其他方法中，并且该 Java 类结构比较简单（可以直接拆分成标量，也即最原始的基本类型）时，则 HotSpot 会将 Java 对象实例直接分配在当前线程所关联的高速缓存中，这种优化策略有一个专门的术语——标量替换。

与方法逃逸类似，如果能够证明一个 Java 对象不会逃逸到其他线程，则该对象便不会存在多线程锁的竞争，那么方法上的同步措施便会消除。很显然，消除了同步锁之后，代码的执行效率会更高。

JVM 中所使用的这些优化技术，每一个都像盛开在阳光下的鲜花，赏之不尽！

而在与 JAVA 关联紧密的另一个世界——安卓虚拟机，也实现了 JIT 技术，但是谷歌的技术大神们觉得这样仍然不够，研究出了一个 AOT（ahead of time）技术。其实所谓 AOT，就是提前编译，或者叫作静态编译，这种技术是相对于 JIT 而言的。AOT 是在 Java 程序运行之前就提前编译好，直接编译成本地相关的机器指令。对于 JVM 而言，如果纯粹使用解释器解释执行，则每次执行一个方法都需要将字节码指令翻译成对应的机器指令，Java 方法调用几次，则这种工作便会被重复做几次；而如果开启 JIT，则每次 Java 程序重新启动运行后，都要进行一次这种编译，这种工作还是重复的。而 AOT 的思想则是，在编译阶段就把这些工作做完，直接将 Java 程序翻译成对应的本地机器指令，这样就不用再在运行期重复去做这些事情了。正是由于 AOT 的出现，安卓原生的 Dalvik 虚拟机便在安卓 5 时代被抛弃了，谷歌转而使用 ART 虚拟机。所谓 ART，其实就是 AOT 的运行时环境，专门负责运行 AOT 后的指令。

上面介绍了 Java/Android 虚拟机的各种优化技术，这些都属于非常底层同时也非常高深的技术范畴，需要非常深厚的技术积累方能玩转。希望国内有兴趣的道友们共同研究，相互分享。

9.6 操作数栈在哪里

JVM 的指令集架构是面向栈的，所设计的大部分字节码指令也都是紧紧围绕栈进行操作，

例如上文提到的 `iadd` 指令，该指令是零地址指令，指令中并没有显式标记其所操作的源数据和目标数据究竟位于哪里，之所以没有显式标记，是因为无论是源数据还是目标数据，其实都位于栈中。讲了半天，这个栈到底在哪里呢，长啥样？大部分书籍中并没有直接的答案。其实这里所谓的栈，是 JVM 内部所实现的一个“求值栈”，这个求值栈也叫作“操作数栈”或者“表达式栈”，JVM 内部将其称为“`expression stack`”。其实前文讲解 JVM 的堆栈实现机制时，已经提到过表达式栈了。当 JVM 准备执行一个 Java 方法时，会先为其创建一个栈帧，前文讲过，栈帧主要包含三大部分，分别是局部变量表、固定帧和操作数栈，其详细结构如图 9.6 所示。

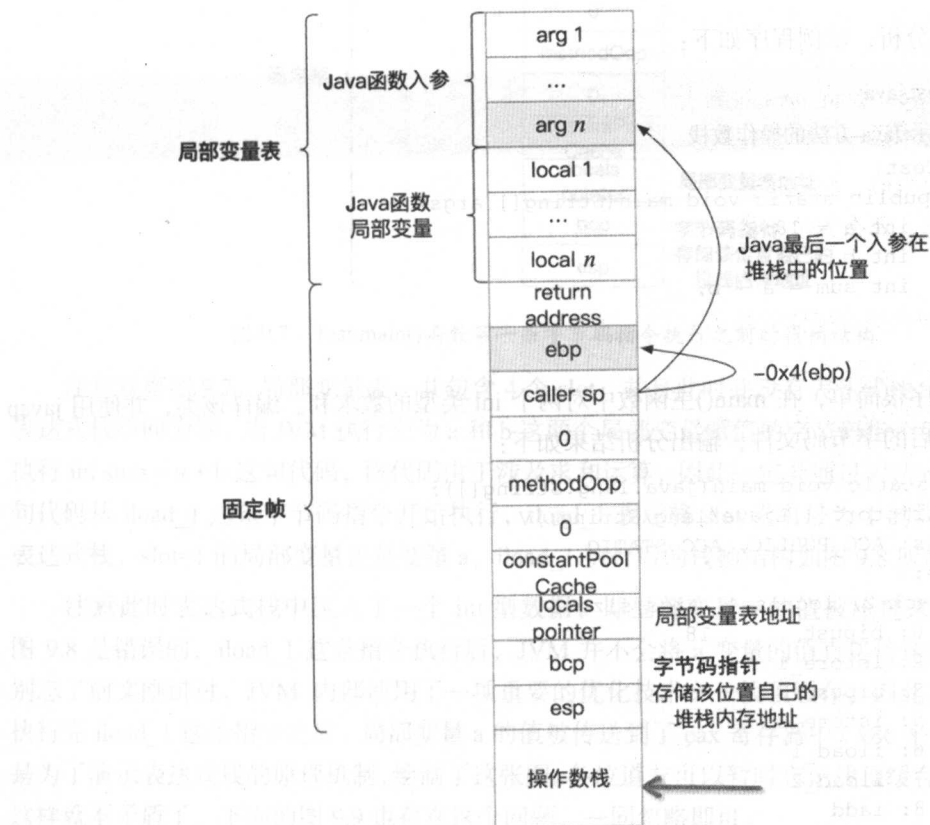


图 9.6 Java 方法栈的三大组成部分

Java 方法栈帧的局部变量表、固定帧和操作数栈按照内存从高位向低位顺序增长（x86 平台），不过在 JVM 开始执行 Java 方法的第一条字节码指令之前，操作数栈其实并没有被创建，JVM 仅仅执行到创建 Java 方法栈帧的最后一步——将当前线程栈栈顶位置压入当前栈顶位置，也即图 9.6 中的最底部操作数栈的上一个存储单元。由于 JVM 在创建 Java 方法栈帧时，将

methodOop、constantPoolCache、bcp（字节码指针）等压入固定帧（fixed frame）中所使用的指令都是 push，因此当执行完之后，物理机器的 esp 指针——栈顶指针，其实就指向当前线程栈的栈顶，这个位置便是图 9.6 中的最底部操作数栈的上一个存储单元。Java 方法所对应的操作数栈便从这个位置开始，因此在 JVM 内部将该位置叫作“expression stack bottom”，即表达式栈栈底。至此，JVM 便为 Java 字节码的执行准备好一切，就等着执行指令。假设 Java 方法的字节码指令中有 iload_1 这条指令，则该指令最终会被翻成本地机器指令 push %eax，该指令会将 Java 方法栈帧的局部变量表中 slot 索引号为 1 的局部变量压入栈顶——从操作数栈底部开始压入。

下面举例分析，示例程序如下：

清单：/Test.java

功能：演示 Java 方法的操作数栈

```
class Test{
    public static void main(String[] args) {
        int a = 18;
        int b = 21;
        int sum = a + b;
    }
}
```

该示例程序很简单，在 main() 主函数中对两个 int 类型的数求和。编译该类，并使用 javap 命令分析编译后的字节码文件，输出分析结果如下：

```
public static void main(java.lang.String[]);
descriptor: ([Ljava/lang/String;)V
flags: ACC_PUBLIC, ACC_STATIC
Code:
    stack=2, locals=4, args_size=1
       0: bipush      18
       2: istore_1
       3: bipush      21
       5: istore_2
       6: iload_1
       7: iload_2
       8: iadd
       9: istore_3
      10: return
```

分析结果显示 locals=4，表示 main() 主函数的局部变量表的 slot 编号最大为 4，同时 args_size=1，表明一共只有 1 个人参。当 JVM 准备调用 main() 主函数的第一个字节码指令时，main() 方法的栈帧结果如图 9.7 所示。

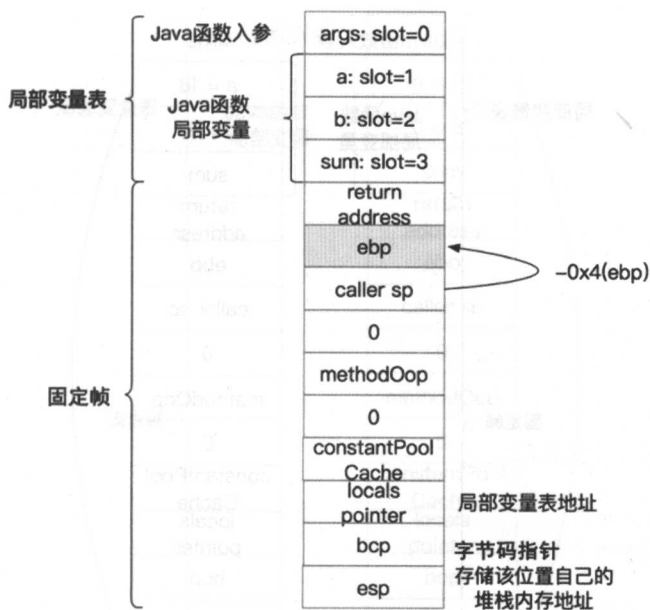


图 9.7 Test.main()函数第一条字节码指令执行之前的栈帧结构

注意观察图 9.7，局部变量表一共包含 4 个 slot，并且此时并没有表达式栈，或者说此时的表达式栈空间为零。当 JVM 执行完为 a 和 b 这两个局部变量赋值的字节码指令后，接着便开始执行 `int sum = a + b` 这句代码，该代码由于涉及求和运算，因此一定会通过表达式栈来完成。这句代码从 `iload_1` 这条字节码指令开始执行，`iload_1` 表示将 slot 索引号为 1 的局部变量传送到表达式栈，slot=1 的局部变量正是变量 a，`iload_1` 执行后的栈帧结构如图 9.8 所示。

注意此时表达式栈中压入了一个 int 型数据，即局部变量 a 的值被压进来。而事实上，图 9.8 是错误的，`iload_1` 这条指令执行后，JVM 并不会将 a 变量的值直接传送到表达式栈中，别忘了前文刚讲过，JVM 内部使用了一项重要的优化技术——栈顶缓存。因此实际的情况是，执行完 `iload_1` 这条指令之后，局部变量 a 的值被传递到了 `eax` 寄存器中（x86 平台）。这里仅仅是为了演示表达式栈的原理机制，绘制了这张图。各位道友可以暂时忘记栈顶缓存技术的存在，这样就不矛盾了。下面的图 9.9 也存在这个问题，一同忽略即可。

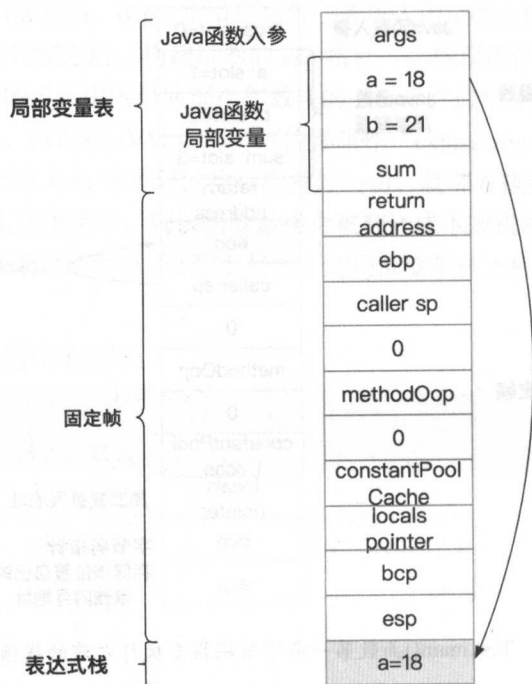


图 9.8 Test.main()函数执行完 iload_1 字节码指令之后的栈帧结构

接着 JVM 执行 `iload_2`, 该字节码指令将 slot 编号为 2 的局部变量的值加载到表达式栈中, 而对于 `Test.main()` 方法而言, slot=2 的局部变量是变量 `b`。该指令执行完之后, `main()` 的方法栈结构如图 9.9 所示。

至此, 表达式栈中已经压入了两个 `int` 型数据, 这就为接下来要执行的 `iadd` 指令准备好了源数据——源数据的确是位于栈顶的。

上面通过一个简单的示例演示了 Java 方法的表达式栈的创建机制, 由此可见, Java 方法的表达式栈其实就位于 Java 方法的栈帧之中。知其然, 更要知其所以然, 方为大善。在这里不禁要问: JVM 为何要将表达式栈放在 Java 方法的栈帧中? 其实答案很简单, 这种方式在技术实现上很简单。假设不这么实现, 而是将表达式栈存放在内存中其他位置, 那么 JVM 得另外维护一套求值栈管理机制了, 比较麻烦。使用这种实现方式, 当创建 Java 方法栈时, 表达式栈的底部位置便已确定; 而当 Java 方法执行完成之后方法栈被销毁时, 表达式栈也会跟着一起被销毁, JVM 无须额外设计一套复杂的机制来管理。

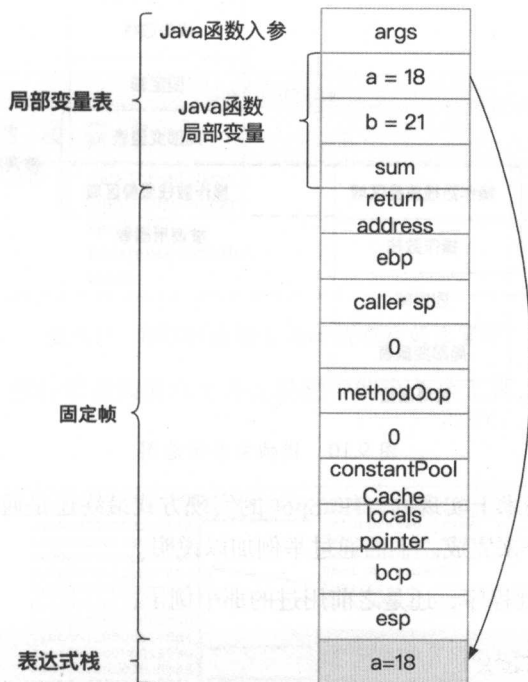


图 9.9 Test.main()函数执行完 iload_2 字节码指令之后的栈帧结构

9.7 栈帧重叠

无论 JVM 的指令集是基于栈还是基于寄存器，其方法调用所基于的数据结构完全相同，都是基于堆栈。前面花了整整一个章节详细地描述了 JVM 内部方法栈帧的创建过程，JVM 在准备调用一个 Java 方法之前，会先为其创建栈帧，随着执行引擎对字节码的执行，JVM 会动态地读写 Java 方法的操作数栈。在概念模型中，存在直接调用关系的两个 Java 方法的栈帧在堆栈空间上是彼此线性串联的，并且彼此都拥有完整的栈帧结构。但是大多数虚拟机的实现都会进行一些优化，其中一项很成熟的优化技术便是栈帧重叠。所谓栈帧重叠，就是使两个相邻的栈帧出现一部分重叠，让前一个栈帧的操作数栈与后一个栈帧的局部变量表区域部分重叠在一起，这样在进行方法调用时就能共用这部分堆栈空间，并且无须进行额外的参数复制。

图 9.10 所示是栈帧重叠的示意图。



图 9.10 栈帧重叠示意图

栈帧重叠是需要技术上实现的。HotSpot 的实现方式最终还是通过 java 字节码，往深了说，最终仍是由机器指令来完成。下面通过举例加以说明。

先准备一个 Java 示例程序，还是之前用过的那个例子：

清单：/Test.java

功能：示例程序

```
public class Test{

    public void add(int a, int b){
        Test test = this;
        int z = a + b;
        int x = 3;
    }

    public static void main(String[] args){
        Test test = new Test();
        test.add(2, 3);
    }
}
```

该示例很简单，在主函数 main() 里面调用 Test 对象实例的 add() 方法。

仅仅讲述理论未免略显枯燥，还是实际练练手来得更加生动。对于堆栈重叠技术，HSDB 依然带着神器的光环，它能够带领各位道友一起领略传说中的堆栈重叠的风姿。

使用 JDB 启动 Test 程序，并在 add() 方法的第 2 行上打上断点，然后就让程序一直处于暂停状态。接着使用 JPS 查看 Test 进程的进程号，使用 HSDB 连接上这个进程，此时 HSDB 的主

窗口如图 9.11 所示。

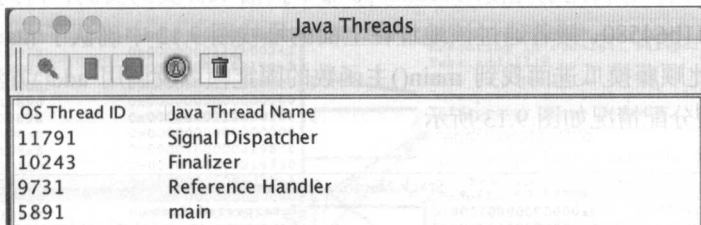


图 9.11 HSDB 连接上 Java 进程后的主窗口

选中 main 主线程，然后单击该窗口上方工具栏中的第 2 个工具，打开 main 主线程的堆栈窗口，如图 9.12 所示。

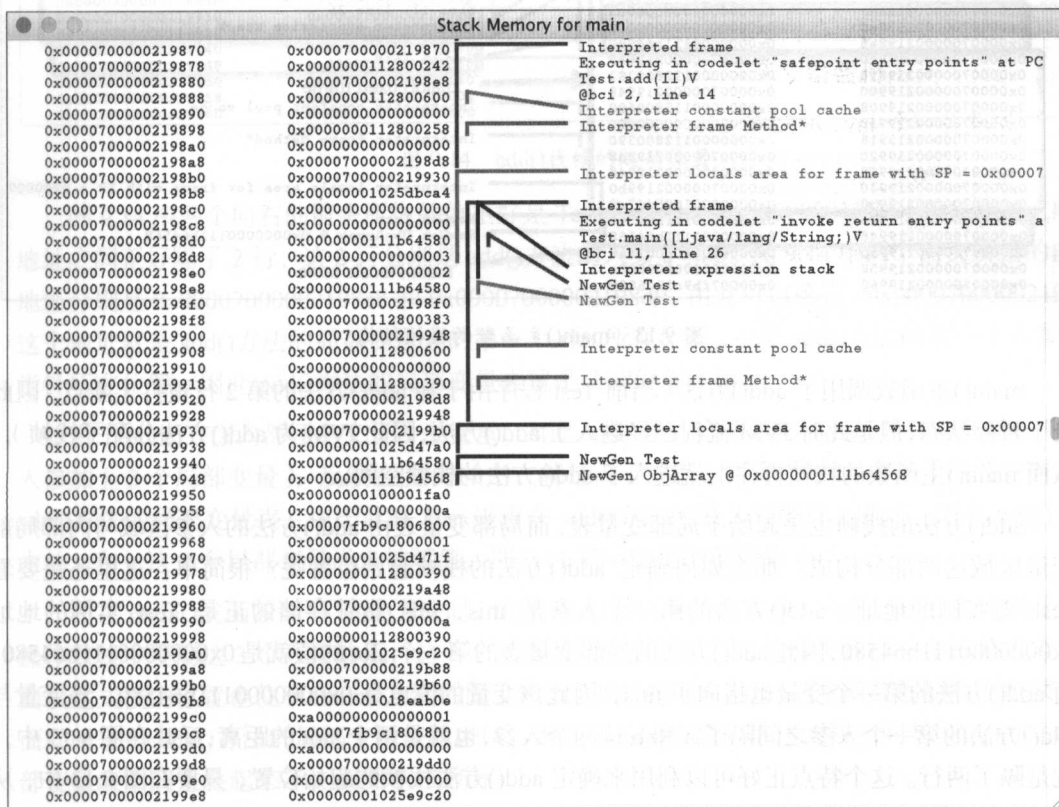


图 9.12 HSDB 显示堆栈详细内容

前面在使用 HSDB 演示 Java 方法栈的一节中其实使用的也是这个例子,并且当时分析了 main()主函数的栈帧。分析之前先使用 scanoops 命令扫描 Test 类的实例,得到 Test 类实例的地址为 0x0000000111b64580,接着通过该地址在上面的堆栈图 9.12 中确认了 main()主函数的栈帧起始位置,并据此顺藤摸瓜进而找到 main()主函数的固定帧以及调用 add()方法时的操作数栈,main()方法的栈帧分配情况如图 9.13 所示。

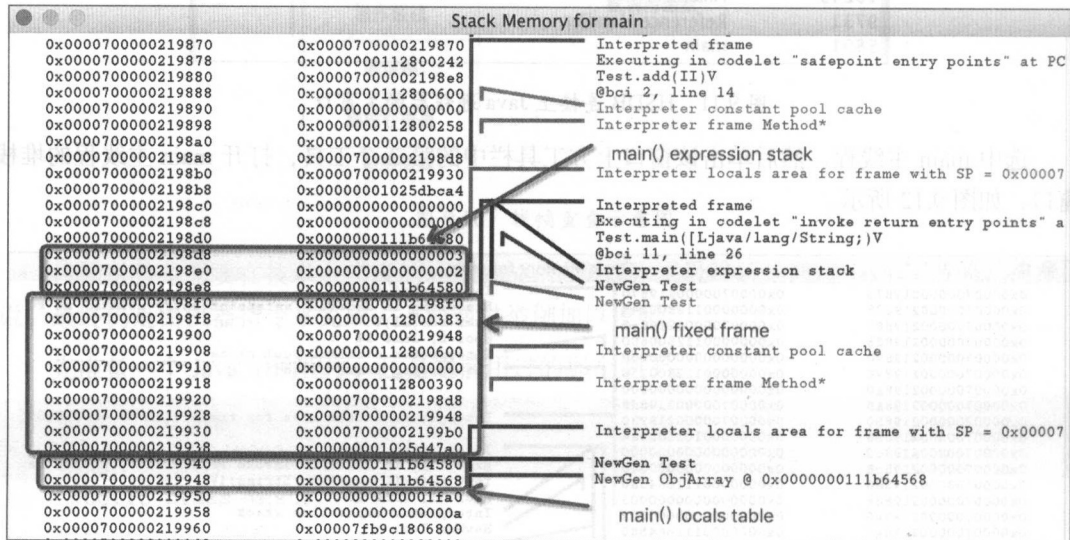


图 9.13 main()主函数的栈帧结构

main()主函数调用了 add()方法(当前 Test 程序由于在 add()方法的第 2 行被打上断点,因此处于暂停状态,但是此时 JVM 流程已经进入了 add()方法,因此 JVM 为 add()方法分配了栈帧),从而 main()主函数的栈帧再往上就进入了 add()方法的栈帧空间。

add()方法的栈帧也是起始于局部变量表,而局部变量表由 add()方法的入参区域与内部局部变量区域这两部分构成。那么如何确定 add()方法的栈帧起始位置呢?很简单,这里还是要看 Test 类实例的地址。add()方法的第一个入参是 this, this 指针存储的正是 Test 实例的地址 0x0000000111b64580,因此 add()方法的局部变量表的第一个 slot 的值就是 0x0000000111b64580。而 add()方法的第一个变量也指向了 this,因此该变量的值也是 0x0000000111b64580。该变量与 add()方法的第一个入参之间隔了 a 和 b 这两个入参,也就是两个 slot 的距离,反映在图 9.13 中,就是隔了两行。这个特点正好可以利用来确定 add()方法栈帧的起始位置,只要在图 9.13 中,从 main()主函数的固定帧区域的顶部往上寻找,找到两处值都是 0x0000000111b64580 并且又隔了两行的地方,就是 add()方法的栈帧的起始位置。从图 9.13 中能够很容易就找到这个位置,如

图 9.14 所示。

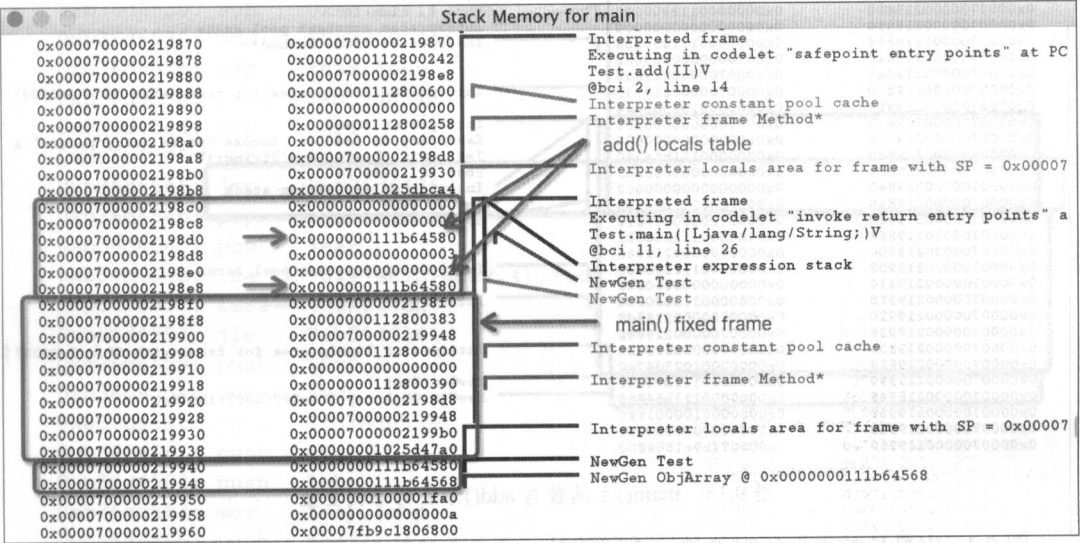


图 9.14 add()栈帧的起始位置

图 9.14 中两个向右的箭头所指的值正好是 Test 实例地址 0x000000011b64580,并且这两个地址之间正好隔了 2 行,这 2 行分别是 add()方法的入参 a 和 b。图 9.14 中两个箭头所指的内存地址分别是 0x00007000002198e8 和 0x00007000002198d0。由此可以确定,0x00007000002198e8 这个地址就是 add()方法的局部变量表的第一个 slot 所在位置,也是 add()方法的第一个入参 this 指针所在位置,因此 add()方法的局部变量表就是从该位置开始。

由于 add()方法内部还定义了 3 个变量,因此 add()方法的局部变量表一共有 6 个成员(3 个人参加上 3 个局部变量),反映在图 9.14 中,就是占了 6 行,因此图 9.14 中最顶部的方框就是 add()方法的局部变量表。由于此时 Test 进程在 add()方法的第 2 行被打上断点,因此 add()方法内的 z 和 x 这 2 个局部变量尚未被赋值,图 9.14 显示它们的值皆为 0。

而在 main()主函数调用 add()方法时,由于 add()方法有 3 个人参,因此 main()主函数需要向操作数栈中复制这 3 个人参的值,这 3 个人参共同组成了运行时的操作数栈(亦称表达式栈),而表达式栈的位置也位于 main()方法的固定帧的上面。而现在,main()主函数栈帧的固定帧的上面同时也是 add()方法的局部变量表的区域,因此 main()方法的表达式栈空间与 add()方法的局部变量表空间重叠起来了,如图 9.15 所示。

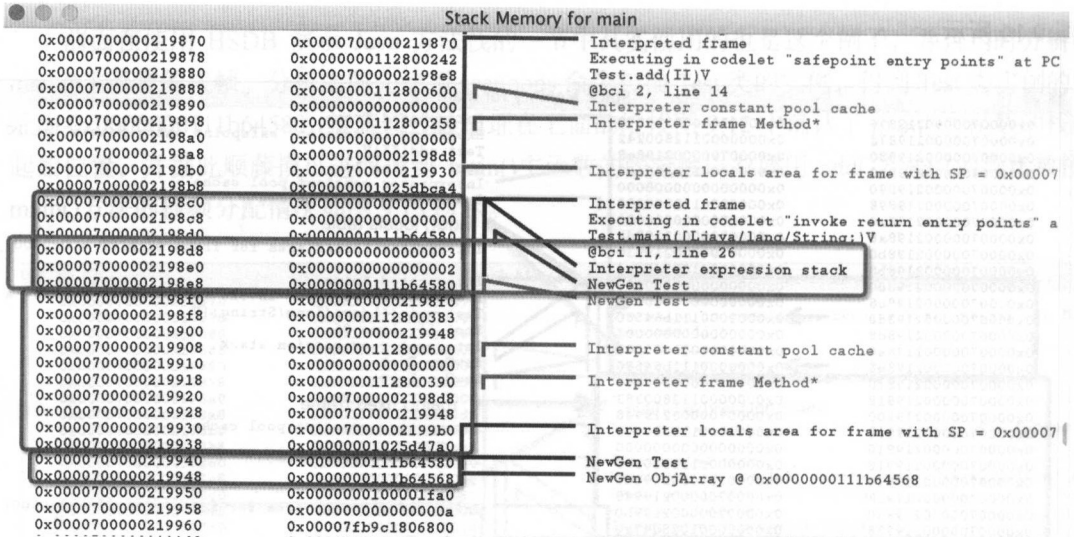


图 9.15 main()主函数与 add()方法的堆栈重叠

图 9.15 中最长的方框所框住的这 3 行就是 main()主函数与 add()方法的栈帧重叠的部分。

由此可以知道，栈帧重叠，所重叠的部分也只是操作数栈部分，不涉及被调用的方法内部的局部变量区域。并且重叠的空间大小，随着当前方法所调用的方法的入参区域大小的不同而不同，并没有固定的大小。各位道友可以自行在自己的电脑上使用 HSDB 来不断实验，进一步加深对栈帧重叠的机制的理解。

9.8 entry_point 例程机器指令

如下：

```
method entry_point (kind = zerolocals) [0xb36d6500, 0xb36d6600] 256 bytes
```

```
[Disassembling for mach='i386']
```

```
0xb36d6500: movzwl 0x26(%ebx),%ecx
0xb36d6504: movzwl 0x24(%ebx),%edx
0xb36d6508: sub    %ecx,%edx
0xb36d650a: cmp    $0x3f6,%edx
0xb36d6510: jbe    0xb36d654c
0xb36d6516: push   %esi
0xb36d6517: mov    %esp,%esi
0xb36d6519: shr    $0xc,%esi
0xb36d651c: mov    -0x48f07d20(,%esi,4),%esi
```

```

0xb36d6523: lea    0x28(,%edx,4),%eax
0xb36d652a: add    0xa8(%esi),%eax
0xb36d6530: sub    0xac(%esi),%eax
0xb36d6536: add    $0x3000,%eax
0xb36d653c: cmp    %eax,%esp
0xb36d653e: ja     0xb36d654b
0xb36d6544: pop    %esi
0xb36d6545: pop    %eax
0xb36d6546: jmp    0xb36d6476
0xb36d654b: pop    %esi
0xb36d654c: pop    %eax
0xb36d654d: lea    -0x4(%esp,%ecx,4),%edi
0xb36d6551: test   %edx,%edx
0xb36d6553: jle    0xb36d6561
0xb36d6559: push   $0x0
0xb36d655e: dec    %edx
0xb36d655f: jg     0xb36d6559
0xb36d6561: push   %eax
0xb36d6562: push   %ebp
0xb36d6563: mov    %esp,%ebp
0xb36d6565: push   %esi
0xb36d6566: push   $0x0
0xb36d656b: mov    0x8(%ebx),%esi
0xb36d656e: lea    0x30(%esi),%esi
0xb36d6571: push   %ebx
0xb36d6572: mov    0x10(%ebx),%edx
0xb36d6575: test   %edx,%edx
0xb36d6577: je     0xb36d6580
0xb36d657d: add    $0x58,%edx
0xb36d6580: push   %edx
0xb36d6581: mov    0xc(%ebx),%edx
0xb36d6584: mov    0xc(%edx),%edx
0xb36d6587: push   %edx
0xb36d6588: push   %edi
0xb36d6589: push   %esi
0xb36d658a: push   $0x0
0xb36d658f: mov    %esp,(%esp)
0xb36d6592: mov    %esp,%eax
0xb36d6594: shr    $0xc,%eax
0xb36d6597: mov    -0x48f07d20(,%eax,4),%eax
0xb36d659e: movb   $0x1,0x175(%eax)
0xb36d65a5: mov    %eax,-0x1000(%esp)
0xb36d65ac: mov    %eax,-0x2000(%esp)
0xb36d65b3: mov    %eax,-0x3000(%esp)
0xb36d65ba: mov    %esp,%eax
0xb36d65bc: shr    $0xc,%eax
0xb36d65bf: mov    -0x48f07d20(,%eax,4),%eax

```

```

0xb36d65c6: movb    $0x0,0x175(%eax)
0xb36d65cd: cmpb    $0x0,0xb70cc4dd
0xb36d65d4: je      0xb36d65f3
0xb36d65da: mov     %esp,%ecx
0xb36d65dc: shr     $0xc,%ecx
0xb36d65df: mov     -0x48f07d20(,%ecx,4),%ecx
0xb36d65e6: mov     -0xc(%ebp),%ebx
0xb36d65e9: push    %ebx
0xb36d65ea: push    %ecx
0xb36d65eb: call    0xb6f330d0
0xb36d65f0: add     $0x8,%esp
0xb36d65f3: movzbl  (%esi),%ebx
0xb36d65f6: jmp     *-0x48f0f2a0(,%ebx,4)
0xb36d65fd: xchg    %ax,%ax

```

9.9 执行引擎实战

JVM 的执行引擎的原理基本分析完了，机智的小伙伴们早就看透了一切！

不过机智是需要表现出来的，如果真的有的话。

而实战是霸气外露的最好机会。阅读 HotSpot 源代码绝不亚于进行一场战争，这场战争的敌军是密密麻麻的源代码。可是，有一块源码在 HotSpot 的源文件中是无法直接看到的，那就是一个 Java 程序最终所生成的本地机器指令，这些指令是 JVM 在运行时动态组装拼接出来的。

这就带来一个很严重的问题，都快要上战场了，结果竟然连敌军在哪里，是谁都不知道，这可是要命的。

本来想在战场上好好秀一秀自己的肌肉，期望在战场上能够打出我军的气势，找出敌军的战略意图，结果找不着敌军，别说霸气侧漏了，根本就露不出来，这仗没法打，这日子没法过啦！

为了让小伙伴们见识一下真正的敌军，笔者也是操碎了心，精心为大家准备了一个示例，为我军引路，找到真正的敌军。

由于在 Java 程序的运行期所动态组装出来的都是本地机器指令，因此如果对汇编实在没兴趣，可以跳过本章，跳过本章并不影响你对 JVM 执行引擎的理解。

9.9.1 一个简单的例子

下面的这个例子十分简单，简单到大家伙儿都瞧不上眼：

清单: /test/A.java

作用: 一个简单的 Java 程序

```
class A{
    public static void doSomething(){
        int a = 3;
        int b = 81;
        int c = a + b - 9;
    }
}
```

使用 `javap` 命令打印这段程序的常量池和方法字节码, 得到如下信息 (仅摘录主要信息):

清单: /A.java

作用: 打印常量池和字节码

Classfile /test/A.class

class A

minor version: 0

major version: 52

flags: ACC_SUPER

Constant pool:

```
#1 = Methodref      #3.#11      // java/lang/Object."<init>":()V
#2 = Class          #12          // A
// ...
```

{

public static void doSomething();

descriptor: ()V

flags: ACC_PUBLIC, ACC_STATIC

Code:

stack=2, locals=3, args_size=0

0: iconst_3

1: istore_0

2: bipush 81

4: istore_1

5: iload_0

6: iload_1

7: iadd

8: bipush 9

10: isub

11: istore_2

12: return

}

根据 `javap` 命令所打印出的 `doSomething()` 方法的信息可知, `doSomething()` 方法的操作数栈最大深度为 2 (`stack=2`), 局部变量表大小为 3 (`locals=3`)。同时, 该方法经过编译后, 一共有

12 条字节码指令。

下面先分析这 12 条字节码指令的运行过程。

9.9.2 字节码运行过程分析

doSomething()方法开始运行之前，JVM 便已经分配好局部变量表和操作数栈（也叫表达式栈，expression stack），具体创建的过程在前文讲解栈帧的章节详细描述过，此处略过不表。对于本例，doSomething()方法的栈帧准备之后的局部变量表与操作数栈内存布局如图 9.16 所示。

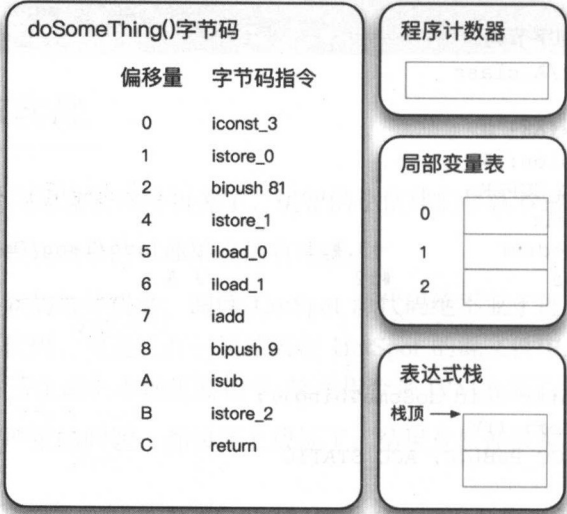


图 9.16 初始堆栈布局

图 9.16 左侧显示字节码，随着程序运行，会标示当前运行到的字节码指令。右侧则显示程序运行过程中的局部变量表和表达式栈的内存数据，同时标注当前程序计数器。

值得注意的是，图 9.16 的左侧分成两列，第一列是当前字节码相对于基址的偏移量，第二列则是具体的字节码。其中第 4 个字节码指令 `istore_1` 的偏移量是 4，而不是 3，这是因为第 3 条字节码指令 `bipush 81` 占用了 2 字节的宽度，`bipush` 占用 1 个，立即数 81 也占用 1 个。同理，第 9 条字节码指令 `isub` 的偏移量是 10，而不是 9，也是因为其上一条指令 `bipush 9` 占用了 2 字节宽度。

所谓的程序计数器，其实上文讲过，就是指示当前字节码指令相对于 Java 方法的字节码区域的起始位置的偏移量所在的堆栈内存位置。对于 x86 的 32 位平台而言，使用 `esi` 寄存器保存

当前字节码指令的内存地址，当前字节码指令运行结束之后，由当前字节码指令自己增加 `esi` 的值，从而将 `esi` 指向下一条字节码指令的内存地址。这一点与 CPU 硬件层面的程序计数器的工作原理类似，只不过 CPU 是纯数字电路驱动，不需要软件程序自己实现计数的逻辑，而 JVM 则由软件逻辑进行控制，但是基于 JVM 这一虚拟机器的上层的 Java 应用程序与基于物理 CPU 之上的软件程序一样，也不需要感知代码指令的偏移。

虽然程序计数器所代表的硬件寄存器中实际所保存的是目标字节码指令的内存地址，但是为了理解简单，大家都不约而同地将其简单理解为指向下一条指令相对于第一条指令的偏移位置，本书也采用这种简化的理解方法，即程序计数器指向的是指令的相对偏移位置。

另外，在 32 位机器上，图 9.16 中局部变量表和表达式栈的每一个小方框代表 4 字节、32 位比特的内存存储单元，前面也分析过，在 32 位平台上，JVM 的一个 slot 槽位正好占据 32 位存储空间。

1. 执行 `iconst_3` 指令

`iconst_3` 指令的作用是将操作数 3 推送至栈顶（表达式栈的栈顶）。执行之后的内存布局如图 9.17 所示。

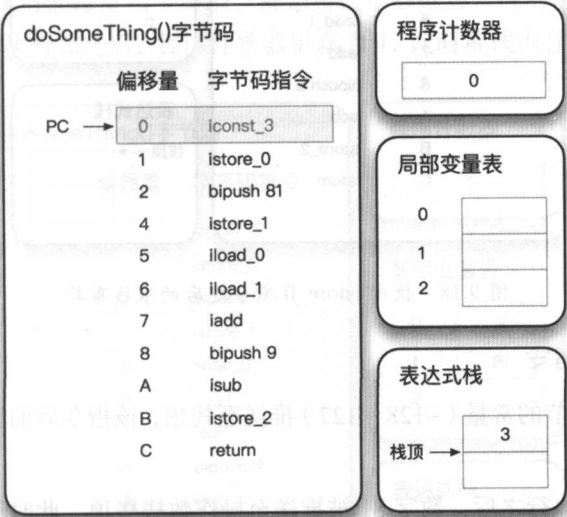


图 9.17 执行 `iconst_3` 指令之后的堆栈布局

此时操作数栈栈顶的值变成 3。根据 JVM 解释器的运行机制，JVM 会先执行字节码指令所对应的逻辑，执行完之后才会将程序计数器更新为下一条字节码指令所在的位置，但是在更新

之前，程序计数器仍然指向当前刚刚执行完的字节码指令。下面为了描述方便，会统一表述为：当执行完当前字节码指令时，程序计数器指向当前字节码指令所在的位置，请细节控不要纠结语言表述上的逻辑问题。

2. 执行 istore_0 指令

istore 指令将表达式栈栈顶的数据弹出来，并传送至局部变量表中指定的位置，该位置由紧跟在 istore 指令后面的数字指定。

当 istore_0 指令执行之后，数字 3 从表达式栈栈顶被弹出，并保存到局部变量表的第 0 个槽位（slot），此时程序计数器的值更新为 1，堆栈内存布局如图 9.18 所示。

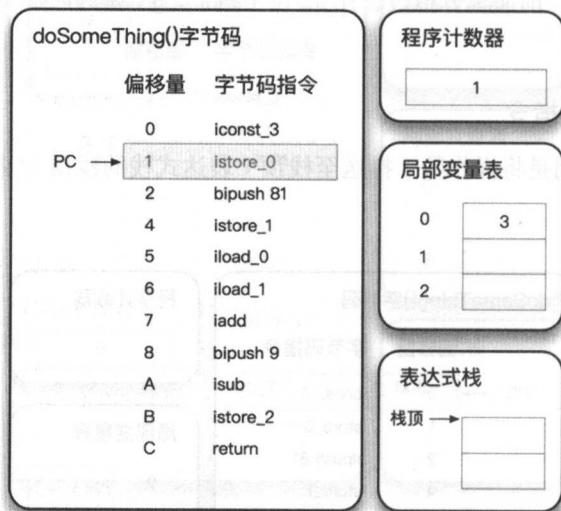


图 9.18 执行 istore_0 指令之后的堆栈布局

3. 执行 bipush 指令

`bipush` 指令将单字节的常量（-128 ~ 127）推送至栈顶，该指令后面紧跟一个单字节的操作数。

当 `bipush 81` 指令执行之后，数字 81 被推送至操作数栈栈顶，此时程序计数器的值更新为 2，堆栈内存布局如图 9.19 所示。



图 9.19 执行 bipush 81 指令之后的堆栈布局

4. 执行 istore_1 指令

istore_1 指令与前面的 istore_0 指令类似，都是将栈顶数据写入局部变量表，不过写入的位置是第 1 个 slot。本指令执行完之后，程序计数器更新为 4，此时堆栈内存布局如图 9.20 所示。

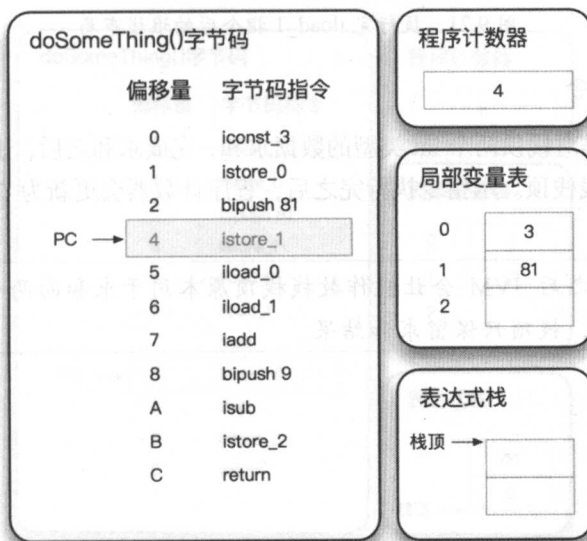


图 9.20 执行 istore_1 指令之后的堆栈布局

5. 执行 `iload_0` 与 `iload_1` 指令

`iload` 系列指令的作用是将局部变量表中的数据推送至栈顶，`iload_0` 与 `iload_1` 指令的作用分别是将局部变量表中 `slot` 索引号为 0 和 1 的两个数据推送至栈顶。在此过程中，程序计数器的值会分别更新为 5 和 6，当执行完 `iload_1` 指令后，堆栈内存布局如图 9.21 所示。

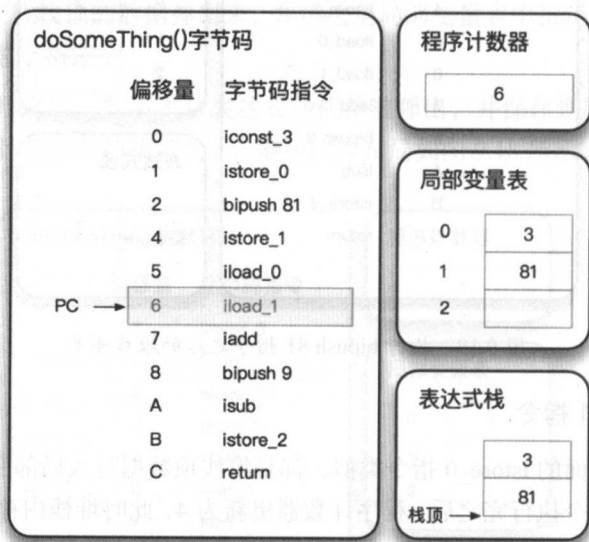


图 9.21 执行完 `iload_1` 指令后的堆栈布局

6. 执行 `iadd` 指令

`iadd` 指令的作用是对栈顶两个 `int` 类型的数据求和，完成求和之后，让栈顶元素出栈，并将累加结果压入操作数栈栈顶。本指令执行完之后，程序计数器会更新为 7，同时堆栈内存布局如图 9.22 所示。

注意，求和之后 JVM 会让操作数栈栈顶原本用于求和的两个数据出栈，所以完成求和之后，栈顶只保留求和结果。

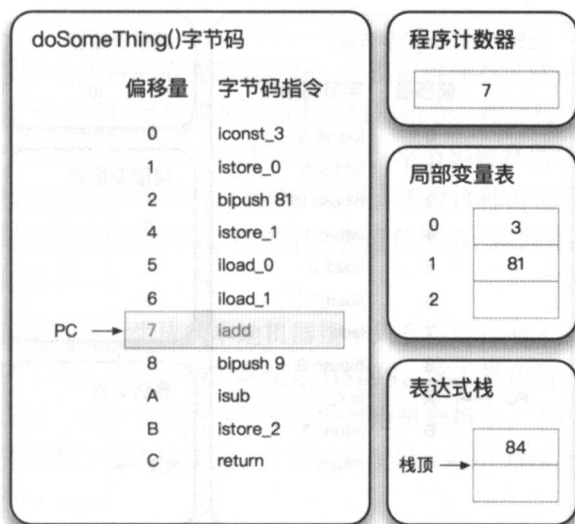


图 9.22 执行完 iadd 指令后的堆栈内存布局

完成 iadd 指令后，后续几条指令是完成减法运算，其中仍会涉及将数据推送至栈顶，执行减法运算，再从栈顶将计算结果写入局部变量表的重复过程，相关指令的流程机制与前面相同，具体过程不再赘述，这里仅贴出后续指令执行过程中的堆栈变化图（如图 9.23、图 9.24 及图 9.25 所示）。

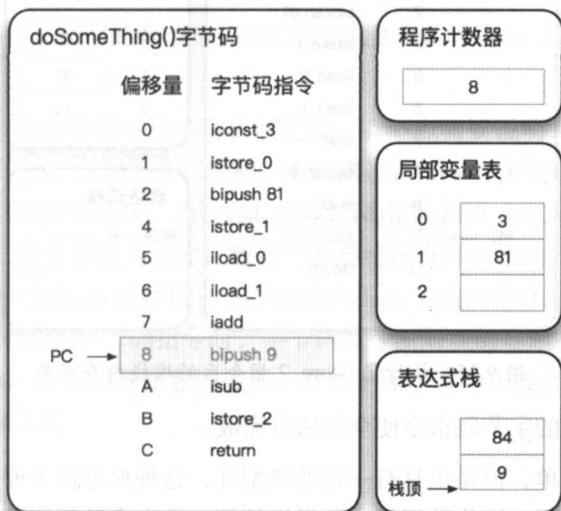


图 9.23 执行完 bipush 9 指令后的堆栈内存布局

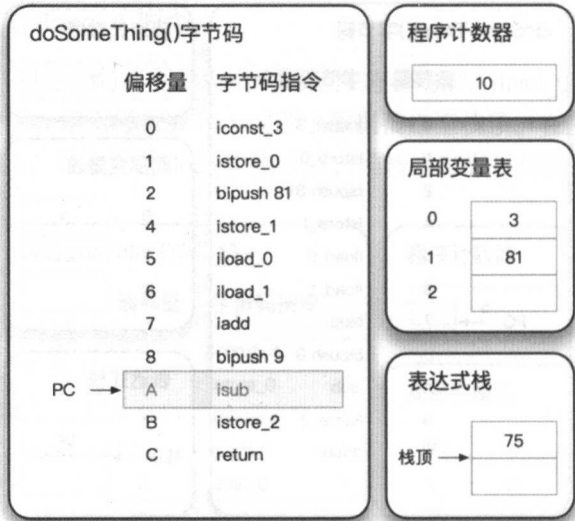


图 9.24 执行完 isub 指令后的堆栈内存布局

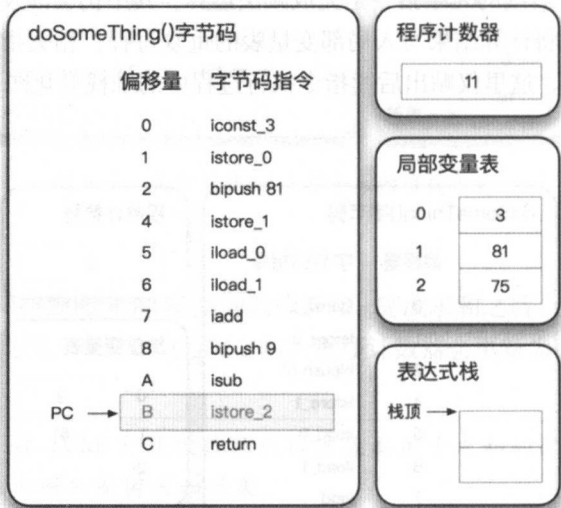


图 9.25 执行完 istore_2 指令后的堆栈内存布局

至此，本示例程序的字节码指令便全部执行完成。

本示例程序虽然简单，但是也具有一定的典型性，这种典型性主要体现在字节码指令上，例如，局部变量表的读写、操作数压栈、数学运算等，绝大多数程序逻辑都离不开这几种最常用的字节码指令。在 JVM 内部，每一种解释器（例如字节码解释器、模板解释器）都给出了

Java 字节码指令对应的实现，或用 C 语言写成，或用机器指令完成。其中字节码解释器的实现最为直观，基本能够看懂 C 语言的道友都能看懂；而模板解释器虽然也使用 C 语言解释字节码逻辑，但是 C 语言仅负责在运行时生成对应的本地机器指令，在运行期实际上是通过动态生成的机器指令去完成字节码指令的逻辑。不过话说回来，字节码解释器使用 C 语言直接解释字节码，虽然编译时会被编译器解释成对应的本地机器码，不过这种由编译器生成的机器指令相比于模板解释器中由人工生成的机器指令，在质量上要逊色很多，这便是字节码解释器从 JVM 很早的版本便被弃用的原因。

不过正是由于模板解释器所生成的本地机器指令只有在运行期才能看到，在编译期无法直接看到，因此必须使用工具，其中一种工具便是 HSDIS，该工具在前文多次提到过。使用该工具能够在运行期打印出每一个字节码指令所对应的本地机器指令，下面便基于本工具，打印本示例程序中常见的字节码指令的本地实现，并分析其逻辑，揭示 Java 字节码指令内部实现的真正原理。

9.10 字节码指令实现

在前文讲解栈顶缓存机制时，提到模板解释器所生成的本地机器指令与栈顶缓存有很大的关系，几乎大部分字节码指令的本地实现都使用了栈顶缓存这种优化技术。这里将要讲解的 `iconst_3`、`istore_0` 和 `iadd` 等指令也都使用了栈顶缓存技术。

同时，在前面讲解栈顶缓存的章节中，其实已经详细讲解了 `iload` 系列指令的本地实现机制，因此这里便不再赘述。

另外，在描述字节码指令的本地实现机制之前，有一点需要说明，当 JVM 开始调用一个 Java 方法之前，会为该 Java 方法创建好栈帧，前面讲过，Java 方法栈帧主要包含 3 大块，分别是局部变量表、固定帧和操作数栈。当 JVM 为即将调用的 Java 方法准备好栈帧之后，在 x86 平台上，JVM 会以 `esi` 寄存器作为程序计数器，并会将该寄存器指向局部变量表的第 0 个 slot 的内存位置，同时 JVM 会将 `sp` 这种栈顶寄存器指向 Java 方法的操作数栈栈顶，因此，在 JVM 执行目标 Java 方法所对应的字节码指令时，字节码指令所对应的本地机器码指令 `push` 与 `pop`，实际上便是在操作 Java 方法的操作数栈栈顶，所以下面所说的压栈和出栈实际上是指对 Java 方法操作数栈的压栈和出栈。

9.10.1 iconst_3

在 32 位 x86 平台上，使用 HSDIS 工具打印该指令对应的本地机器逻辑如下：

```
iconst_3 6 iconst_3 [0xb36d7ce0, 0xb36d7d20] 64 bytes

[Disassembling for mach='i386']
0xb36d7ce0: sub    $0x4,%esp
0xb36d7ce3: fstps  (%esp)
0xb36d7ce6: jmp    0xb36d7d04
0xb36d7ceb: sub    $0x8,%esp
0xb36d7cee: fstpl  (%esp)
0xb36d7cf1: jmp    0xb36d7d04
0xb36d7cf6: push   %edx
0xb36d7cf7: push   %eax
0xb36d7cf8: jmp    0xb36d7d04
0xb36d7cfd: push   %eax
0xb36d7cfe: jmp    0xb36d7d04
0xb36d7d03: push   %eax ----->栈顶缓存入口点
0xb36d7d04: mov    $0x3,%eax ----->无缓存入口点
0xb36d7d09: movzbl 0x1(%esi),%ebx ----->取指逻辑开始
0xb36d7d0d: inc     %esi
0xb36d7d0e: jmp    *-0x48f106a0(,%ebx,4)
0xb36d7d15: xchg    %ax,%ax
0xb36d7d18: add     %al,(%eax)
0xb36d7d1a: add     %al,(%eax)
0xb36d7d1c: add     %al,(%eax)
0xb36d7d1e: add     %al,(%eax)
```

`iconst_3` 指令对应的本地机器码貌似很多，但是对于本示例程序（即 9.9 节“执行引擎实战”中的简单 Java 测试程序，下同）而言，由于 `iconst_3` 指令是方法的第一条字节码指令，在其之前并没有其他字节码指令会向栈顶缓存数据，因此此时栈顶状态为空，所以 `iconst_3` 指令便从上面这段本地机器指令的“无缓存入口点”进入，即下面这条机器指令：

```
mov $0x3, %eax
```

该指令驱动物理 CPU 将操作数 3 传送到 `eax` 寄存器中。JVM 完成这条传送指令之后，便直接开始取指，准备执行下一条字节码指令。这中间似乎存在一个问题，不是说好的，`iconst_3` 指令会将数字 3 压入操作数栈栈顶的吗，为何这里仅仅看到机器指令将其传送到寄存器中，反倒没栈顶啥事儿了？答案很简单，这里仍然在履行栈顶缓存策略，`iconst_3` 指令并没有真的将数据入栈，而是先临时存放在 `eax` 这个缓冲器中。

那么 `iconst_3` 指令到底啥时候才会将数字 3 入栈呢？这需要看具体的场景，具体来说需要看其后面的那条字节码指令是啥，如果其后面的那条指令仍然是进行压栈操作，例如又来一条

iconst 系列的指令，或者来一条 iload 系列的指令，或者 sipush、bipush、ldc 等指令，由于这些指令也会使用栈顶缓存策略，而对于一个给定的 CPU 硬件平台，栈顶缓存只能有一个寄存器，所以 JVM 为了给这些后续的指令腾出栈顶缓存空间，只能对 iconst_3 指令中所隐含的操作数 3 执行真正压栈操作。但是如果 iconst_3 指令后面的那条字节码指令不是压栈操作，而是运算指令，例如 iadd、isub 等，或者是写局部变量表操作，例如 istore 系列的指令等，则 JVM 永远不会对 iconst_3 执行真正的压栈操作，数字 3 最多只能到达用于栈顶缓存的寄存器之中，而不会被传送到栈顶，这是为了减少几次内存读写操作，从而提升运算速度。

对于本示例程序，由于 iconst_3 指令后面的那条指令是 istore_0，该指令不会继续进行压栈操作，因此实际上 JVM 在运行本测试程序时，并不会将数字 3 压入栈顶，从这个角度而言，其实上一节所绘制的内存堆栈布局图是错误的，不过上一节的重点是分析字节码的字面含义，并不考虑栈顶缓存这种优化技术，因此基于字节码字面含义而绘制的堆栈布局图并没有问题。事实上，JVM 规范也只是定义出了一套字节码指令集，至于各种 JVM 虚拟机内部究竟如何实现，则没有相应的规范，所以通常对字节码指令的理解也只能是基于其字面含义。

9.10.2 istore_0

在 32 位 x86 平台上，使用 HSDIS 工具打印该指令对应的本地机器逻辑如下：

```
istore_0 59 istore_0 [0xb36d9240, 0xb36d9260] 32 bytes
```

```
[Disassembling for mach='i386']
```

```
0xb36d9240: pop    %eax ----->无缓存入口点
```

```
0xb36d9241: mov    %eax, (%edi) ----->栈顶缓存入口点
```

```
0xb36d9243: movzbl 0x1(%esi), %ebx ----->取指逻辑开始
```

```
0xb36d9247: inc    %esi
```

```
0xb36d9248: jmp    *-0x48f0f2a0(, %ebx, 4)
```

```
0xb36d924f: nop
```

```
0xb36d9250: add    %al, (%eax)
```

```
0xb36d9252: add    %al, (%eax)
```

```
0xb36d9254: add    %al, (%eax)
```

```
0xb36d9256: add    %al, (%eax)
```

```
0xb36d9258: add    %al, (%eax)
```

```
0xb36d925a: add    %al, (%eax)
```

```
0xb36d925c: add    %al, (%eax)
```

```
0xb36d925e: add    %al, (%eax)
```

本示例程序所对应的第二条字节码指令是 istore_0，该指令的字面含义是将栈顶数据写入局部变量表中索引号为 0 的 slot 槽位中。

同样，本字节码指令依然使用了栈顶缓存技术，其对应的第一条机器指令是 pop %eax，这

条机器指令的含义是将栈顶数据弹出至 `eax` 寄存器中。

在 JVM 执行字节码跳转时，会判断栈顶缓存状态，当栈顶缓存为空时，则会执行 `pop` 系列的机器指令以将栈顶数据弹出至用于缓存的寄存器之中。这是一个通用逻辑。在 `x86` 平台上，用作栈顶缓存的寄存器是 `eax`，在 JVM 执行 `istore` 系列的字节码指令之前，如果 `eax` 寄存器中已经有数据，则 JVM 不再执行 `pop %eax` 这条机器指令，而是直接从该指令的下一条指令——`mov %eax, (%edi)` 开始执行。这条指令将 `eax` 寄存器中的数据传送到 `(%edi)` 所指向的内存位置，前面讲过，`edi` 寄存器指向 Java 方法栈帧的局部变量表第 0 个 slot 位置，因此这条指令实际上在解释执行 Java 字节码指令 `istore` 的字面含义——将栈顶数据写入局部变量表。

9.10.3 iadd

在 32 位 `x86` 平台上，使用 `HSDIS` 工具打印该指令对应的本地机器逻辑如下：

```
iadd 96 iadd [0xb36d9f40, 0xb36d9f60] 32 bytes

[Disassembling for mach='i386']
0xb36d9f40: pop    %eax ----->无缓存入口点
0xb36d9f41: pop    %edx ----->缓存入口点
0xb36d9f42: add    %edx, %eax
0xb36d9f44: movzbl 0x1(%esi), %ebx ----->取指逻辑开始
0xb36d9f48: inc    %esi
0xb36d9f49: jmp    *-0x48f106a0(, %ebx, 4)
0xb36d9f50: add    %al, (%eax)
0xb36d9f52: add    %al, (%eax)
0xb36d9f54: add    %al, (%eax)
0xb36d9f56: add    %al, (%eax)
0xb36d9f58: add    %al, (%eax)
0xb36d9f5a: add    %al, (%eax)
0xb36d9f5c: add    %al, (%eax)
0xb36d9f5e: add    %al, (%eax)
```

`iadd` 指令的作用是对 Java 方法栈栈顶的两个 `int` 型数据执行求和运算。由于 JVM 本身不具备数学运算的能力，最终仍然要依靠物理 CPU 才能完成，而在 `x86` 平台上，物理机器执行求和运算的一种方式便是直接对两个寄存器中的数据进行累加，例如：

```
add %edx, %eax
```

JVM 执行求和逻辑时，也会将 Java 方法栈栈顶的两个数据传送到 `edx` 和 `eax` 这两个寄存器中，这样才能触发 CPU 硬件的求和指令，从而完成真正的求和运算。

`iadd` 指令同样履行了栈顶缓存策略，如果缓存寄存器 `eax` 中没有数据，则会从上面第一条

机器指令 `pop %eax` 开始执行, 将 Java 方法栈栈顶的第一个 `int` 型数据弹出至 `eax` 寄存器中, 接着执行上面第二条机器指令 `pop %edx`, 将 Java 方法栈栈顶的第二个 `int` 型数据弹出至 `edx` 寄存器中。而如果缓存寄存器 `eax` 中已经有数据, 例如 `iadd` 指令的上一条指令是 `iload` 系列或者 `iconst` 系列的指令等, 这些指令会将操作数缓存至 `eax` 寄存器中, 因此在执行 `iadd` 指令时, 会直接从上面第二条机器指令开始执行, 第一条机器指令不需要执行。如此一来, 原本需要连续执行两次 `pop` 指令才能将栈顶的两个数据弹出至寄存器中, 现在只需要执行一次 `pop` 指令。CPU 在执行 `pop` 指令时需要将数据从内存传送至寄存器中, 而读写内存的效率相比于读写寄存器是非常低的, 因此 JVM 每节省一次内存读写, 性能便能提高不少。

总体而言, JVM 虽然有一个专门的执行引擎模块, 能够执行常规的若干指令, 但是毕竟计算机的运算能力只能依靠硬件赋予, 因此 JVM 字节码指令最终都需要转换为对应的硬件 CPU 指令。上面展示了在 x86 平台上的几种常见的字节码指令的解释原理, 其他字节码指令的解释原理也都大同小异, 都是基于 Java 方法操作数栈和局部变量表而翻译成对应的机器逻辑。

9.11 本章总结

JVM 最核心的技术便是执行引擎, 最难的也是执行引擎, 而这也是本书写作的初衷!

要想透彻理解 JVM 的执行引擎, 就必须先理解物理计算机 CPU 执行运算的机制。本书详细描述了物理 CPU 进行取指、译码、运算的原理, 并从这个点出发, 逐步深入讲解 JVM 的执行引擎的运行机制。

相比于物理 CPU 的取指机制, JVM 的取指机制显得更加复杂。从整体效果来看, JVM 的取指机制其实糅合了物理 CPU 的取指机制和 JVM 本身的字节码取指机制。由于一个 Java 方法对应若干字节码指令, 因此每当 JVM 执行完一条字节码指令后, 便需要执行一次“取指”。而每一条字节码指令又对应多条机器指令, 因此在 JVM 执行一条目标字节码指令时, 需要同时处理本地机器指令的跳转。

JVM 在执行字节码指令时, 综合使用了若干技巧, 这些技巧可以节省 CPU 资源, 提高程序性能。这些技巧包括栈顶缓存、堆栈重叠及 JIT 等。其中 JIT 属于非常高级的技术主题, 同时也是一个永恒的话题, 毕竟既想拥有 Java 简单易学的语法特性(最重要的是不需要管理内存), 又想尽可能地提升程序执行效率, 不是一件容易的事。

第 10 章

类的生命周期

本章摘要

- ◎ 类的生命周期
- ◎ 类加载的内部实现及触发
- ◎ 类的初始化
- ◎ 类加载器的本质
- ◎ 类实例分配

Java 类的生命周期是一个绕不开的话题，本书将从源代码的角度来讲解 Java 类生命周期的技术实现细节。上一章所描述的 Java 执行引擎，更是与类的生命周期息息相关——Java 执行引擎直接负责和管理 Java 类生命周期的大部分阶段，包括类的加载、初始化、创建与方法调用。

10.1 类的生命周期概述

Java 程序的所有数据结构和算法都封装在类型之中，这也是面向对象编程语言的一大特色。当 JVM 执行一个 Java 类所封装的算法之前，首先要做的一件事便是字节码文件解析，字节码文件解析包含 3 个主要的过程——常量池解析、Java 类字段解析及 Java 方法解析。通过类字段解析，JVM 能够分析出 Java 类所封装的数据结构；通过方法解析，JVM 能够分析出 Java 类所封装的算法逻辑。而无论是数据结构还是方法信息，很多与“字符串”或者大数据（是指占二进制位比较多的大数）相关的信息都封装于常量池中，所以 JVM 欲解析字段和方法信息，必先解析常量池。前文详细描述了字节码文件解析的技术实现细节，当常量池、字段和方法信息全部

被解析完，则字节码文件的“精华”便已经被完全消化吸收。但是，这几个过程其实仅仅属于 Java 类“加载”过程中的一个环节，这对于一个 Java 类的整个“漫长”的生命周期而言，仅仅是个开始。在字节码文件的精华被吸收之后还需要经过一系列的“二次”消化处理，方能被 JVM 在运行期“随心所欲”地调用。

按照 JVM 规范，一个 Java 文件从被加载到被卸载的整个生命过程，总共要经历 5 个阶段：加载→链接（验证+准备+解析）→初始化（使用前的准备）→使用→卸载。其中第二个阶段“链接”，对应了 3 个阶段：验证、准备和解析，因此，也有很多典籍说 Java 类的生命周期一共包括 7 个阶段。

前文所讲的常量池解析、Java 字段和方法的解析，其实都属于加载阶段的一部分。所谓加载，简而言之就是将 Java 类的字节码文件加载到机器内存中，并在内存中构建出 Java 类的原型——类模板对象。所谓类模板对象，其实就是 Java 类在 JVM 内存中的一个快照，JVM 将从字节码文件中解析出的常量池、类字段、类方法等信息存储到类模板中，这样 JVM 在运行期便能通过类模板而获取 Java 类中的任意信息，能够对 Java 类的成员变量进行遍历，也能进行 Java 方法的调用。反射的机制即基于这一基础。如果 JVM 没有将 Java 类的声明信息存储起来，则 JVM 在运行期也无法反射。字节码相关的工具类库，例如 asm、cglib 等，都利用了这一机制，在运行期动态修改静态声明的 Java 类所对应的字节码内容，从而在运行期直接改掉 Java 类的定义，甚至直接在运行期创建一个全新的 Java 类。

Java 类是写给人类看的，而 JVM 内存中的类模板快照则是写给机器“看”的。物理机器无法直接执行 Java 类的源代码，所以需要通过类加载这样一个过程将字节码格式的 Java 类转换成机器能够识别的内存类模板快照。

JVM 完成 Java 类加载之后，接着便开始进行链接。所谓链接，虽然与编译原理中的链接不是同一件事，然而本质上是相同的。总体而言，链接的主要作用是将字节码指令中对常量池中的索引引用转换为直接引用。链接包含 3 个步骤：验证、准备和解析。其实在类加载阶段（也即类的生命周期的第一个阶段），JVM 会对字节码文件进行验证，只不过该阶段的验证着重于字节码文件格式本身，与“链接”阶段的验证侧重点不同。在链接阶段，着重于由字节码信息出发进行反向验证，例如，验证根据字节码文件中的类名是否能够找到对应的类模板。这些都验证无误之后，JVM 才能放心地加载当前类，也才能放心地将字节码指令中对常量池索引号的引用重写为直接引用。关于链接的具体技术实现，实在是太重要了，因为在 JVM 执行字节码指令的过程中，会依赖于重写机制，例如 `invokevirtual` 指令。重写本身比较简单，但是与方法调用联系到一起，就比较复杂了，本书暂时不对此进行深入分析，不知各位道友的胃口深浅，如果各位的口味都比较“重”，那么下一版中要好好讲一讲。

在链接阶段，在正式使用 Java 类之前的最后一道工序便是“初始化”。这里所谓的初始化，

并非指对类进行实例化，而是指执行类的<clinit>()方法。Java 类的实例化，对应的乃是 Java 类生命周期中的“使用”阶段。类的<clinit>()方法在前文讲解 JVM 解析 Java 类方法时曾经详细分析过，若有不清楚的小伙伴可以翻看那一章。总体而言，当 Java 类中包含 static 修饰的静态字段，或者有使用 static {} 块包裹的代码段时，编译后便会在字节码文件中包含一个名为<clinit>()的方法，JVM 在初始化阶段便会调用该方法。需要说明一点，该方法仅能由 Java 编译器生成并由 JVM 调用，程序开发者无法自定义一个同名的方法，更无法直接在 Java 程序中调用该方法，虽然该方法也是由字节码指令所组成。

等 JVM 完成类的初始化之后，便“万事俱备，只欠东风”，就等着开发者使用了。使用方式多种多样，其中最常见的一种方式是通过 new 关键字来实例化一个 Java 类。当然，除了通过 new 关键字使用 Java 类，还可以有多种方式，例如下面这个例子：

清单：/Test.java

功能：演示 Java 类的使用

```
public static void main(String[] args) throws Exception {
    Map map = (Map)Class.forName("java.util.HashMap").newInstance();
    map.put("aaa", "sss");
    System.out.println("map: " + map.get("aaa"));
}
```

该示例使用 Class.forName(String)接口加载一个类，并通过 Class.newInstance()接口实例化一个类。

从广义上说，类的加载也可以对应类生命周期的 7 个阶段中的前 5 个阶段，即加载、验证、准备、解析和初始化。当类加载之后，JVM 内部会为 Java 类创建一个对等的类模板，类模板在 JDK 6 时代被存储在所谓的 perm 区，而到了 JDK 8 时代，则被存储在所谓的 metaSpace 区。无论存储在哪里，当存储区即将被打爆而这个类又不再使用时，JVM 的 GC 便有可能将其回收，即释放内存。而当实例化一个 Java 类之后，JVM 内部则会为 Java 类实例对象创建一个对等的实例对象，该实例对象所存储的区域与具体的 GC 策略紧密关联，有可能在新生代，也可能在老年代，当然，更可能在栈上（栈上分配）。当类被使用完毕之后，JVM 必须销毁实例对象，否则 JVM 内存区早晚会被打爆。JVM 对类模板的销毁和类实例对象的销毁，都是卸载。

总体而言，Java 类的生命周期如图 10.1 所示。

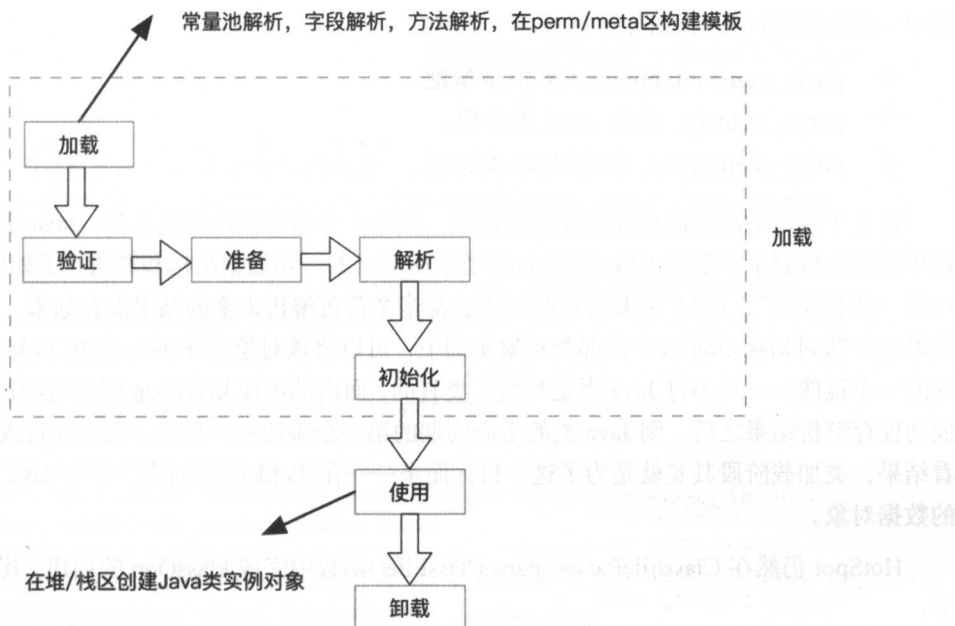


图 10.1 Java 类的生命周期

虽然 Java 类的生命周期包含 7 个阶段，然而这 7 个阶段到底做了些什么事情，内部实现机制如何，我们仍然一无所知。如果不了解这些内部机制，则即使知道这 7 个阶段，怕是也没啥用处。所以下面还是通过 HotSpot 来一窥其中的门道。

10.2 类加载

前文已经描述过 JVM 对字节码文件的精华部分的解析过程，当字节码文件解析完成之后，JVM 便会在内部创建一个与 Java 类对等的类模板对象，说白了该对象其实是 C++ 类的实例。每一个 Java 类模型，最终在 JVM 内部都会有一个 `klassOop` 与之对等，Java 类中的字段、方法及常量池等都会保存到 `klassOop` 实例对象中。要注意，这个实例对象并非 Java 类的实例对象，其仅仅用于表示 Java 类型本身，或者 Java 类的定义。与 Java 类实例对象对等的 JVM 内部对象是 `instanceOop` 实例。

下面就从 Java 类模板对象——`instanceKlass` 的创建开始讲起。

前面描述过 Java 字节码文件的常量池解析、字段解析与方法解析，这三部分内容的解析便是 Java 字节码文件的精华所在。这三部分内容的解析都位于 `ClassFileParser::parseClassFile()` 函

数中，对应的接口分别如下：

- ◎ `parse_constant_pool()`，解析常量池。
- ◎ `parse_fields()`，解析 Java 类字段。
- ◎ `parse_methods()`，解析 Java 类方法。

这 3 个接口的源代码在前面详细解读过。当这 3 个接口执行完成之后，Java 字节码文件的精华便被分析完了，至此 JVM 便对 Java 类中所定义的一切数据结构和算法“了如指掌”，为了巩固“胜利成果”，JVM 需要将这些好不容易辛辛苦苦解析出来的结果保存起来。这些解析的结果会存储到 `klassOop` 这个内部类对象实例中，可以将该对象看作 Java 类在 JVM 内部完全对等的一个镜像——只不过 Java 类是写给人类看的，而内部镜像 `klassOop` 则是写给机器读的。当成功保存解析结果之后，则 Java 类的生命周期的第一个阶段——加载，便大功告成。不看过程看结果，类加载阶段其实就是为了这一目标而来——在 JVM 内部创建一个与 Java 类结构对等的对象。

HotSpot 仍然在 `ClassFileParser::parseClassFile()` 函数中完成 `klassOop` 的创建，其主要源码如下：

清单：/src/share/vm/classfile/classFileParser.cpp

功能：创建 `klassOop`

```
instanceKlassHandle ClassFileParser::parseClassFile(Symbol* name,
                                                    Handle class_loader,
                                                    Handle protection_domain,
                                                    KlassHandle host_klass,
                                                    GrowableArray<Handle*> cp_patches,
                                                    TempNewSymbol& parsed_name,
                                                    bool verify,
                                                    TRAPS) {
    // ...
    // 解析常量池
    constantPoolHandle cp = parse_constant_pool(CHECK_(nullHandle));

    //.....
    // 解析 Java 类中定义的字段
    typeArrayHandle fields = parse_fields(cp, access_flags.is_interface(),
    &fac, &fields_annotations, CHECK_(nullHandle));

    // ...
    // 解析 Java 类中定义的方法
    objArrayHandle methods = parse_methods(cp, access_flags.is_interface(),
    &promoted_flags,
    &has_final_method,
    &methods_annotations_oop,
```

```

        &methods_parameter_annotations_oop,
        &methods_default_annotations_oop,
        CHECK_(nullHandle));

// ...
// 开始创建与 Java 类对等的内部对象
klassOop ik = oopFactory::new_instanceKlass(vtable_size, itable_size,
                                             static_field_size,
                                             total_oop_map_count,
                                             rt, CHECK_(nullHandle));

instanceKlassHandle this_class (THREAD, ik);

// ...
// Fill in information already parsed
this_class->set_access_flags(access_flags);
this_class->set_should_verify_class(verify);
jint lh = Klass::instance_layout_helper(instance_size, false);
this_class->set_layout_helper(lh);
assert(this_class->oop_is_instance(), "layout is correct");
assert(this_class->size_helper() == instance_size, "correct size_helper");
this_class->set_class_loader(class_loader());
this_class->set_nonstatic_field_size(nonstatic_field_size);
this_class->set_has_nonstatic_fields(has_nonstatic_fields);
// ...

// 设置 itable 偏移表
klassItable::setup_itable_offset_table(this_class);
}

```

在这段源码中，笔者将常量池解析、Java 类字段解析、Java 方法解析的调用同时贴了出来，这样方便各位道友把握字节码文件解析的主脉络。在创建 `klassOop` 时，首先通过调用 `oopFactory::new_instanceKlass()` 接口在内存中构建一个 `instanceKlass` 对象实例，该接口的机制与在前文介绍过的 JVM 构建常量池对象实例的逻辑类似，如果有不熟悉的小伙伴可以专门看看常量池解析的那一章，这里不再赘述。不过与常量池的构建、方法对象 `methodOop` 的构建一样，构建实例对象绕不开的一个问题是，所创建的对象占多大的内存空间？要弄清楚这个问题，很简单，只需要看下在调用 `oopFactory::new_instanceKlass()` 接口时所传入的参数，上面的源码包含了该接口调用的部分，从上面的源码可知，在调用该接口时传入了 `vtable_size`、`itable_size`、`static_field_size` 和 `total_oop_map_count` 这 4 个与大小有关的数据，之所以要传入这 4 个数据，是因为它们与 `klassOop` 在内存中的实际布局是有关系的。`oopFactory::new_instanceKlass()` 接口所构建的对象类型是 `instanceKlass`，该类型继承自 `Klass` 类，其内部结构如下：

```

instanceKlass//类结构
    Klass//结构部分
        jint    _layout_helper    //布局类型

```

```

juint    _super_check_offset
Symbol*   _name        //类名
klassOop  _secondary_super_cache
// ...
jint      _biased_lock_revocation_count
instanceKlass//结构部分
klassOop   _array_klasses
objArrayOop  _methods    //Java 类中定义的方法信息
typeArrayOop _fields     //Java 类中定义的字段信息
oop         _class_loader //类加载器
typeArrayOop _inner_classes //内部类
int         _nonstatic_field_size //非静态字段的大小
int         _static_field_size   //静态字段大小
int         _vtable_len         //虚方法表长度
// ...
volatile u2 _idnum_allocated_count

```

从 instanceKlass 的结构可以看到，其内部定义了若干字段，这些字段足以存储 Java 类规范所支持的一切信息，例如字段、方法、内部类等，因为 instanceKlass 要作为 Java 类在 JVM 内部对等的结构体，所以能够兼容 Java 类中的所有元素是其唯一的设计目标。但是 JVM 在创建 instanceKlass 对象时，为其所申请的内存空间却超过了 instanceKlass 类型本身所需的内存大小，这是因为 JVM 需要在 instanceKlass 内存空间的末尾再预留出足够的空间，存储虚方法表 vtable、接口表 itable 及 JAVA 类中的引用类型表 oopMap。存储虚方法表 vtable，其作用在前文分析 Java 方法的解析机制时详细描述过，这里不再赘述。itable 与 oopMap 也是各有其作用。不过在调用 oopFactory::new_instanceKlass() 接口创建 instanceKlass 对象实例时，还传入了 static_field_size 这个数据，其表示 Java 类中所定义的静态字段所占内存的大小。不过静态字段在不同的 JDK 版本中存放的位置不同，在 JDK 6 中，静态字段会被分配到 instanceKlass 实例对象所申请的内存空间中，而在 JDK 7 和 8 中，静态字段将会被分配到与 instanceKlass 对等的镜像类——java.lang.Class 实例中，关于静态字段的分配及镜像类会在下文详细分析，此处先略过不表。由于在 JDK6 中，静态字段信息也会存放在 instanceKlass 对象的预留内存空间中，因此最终 JVM 为 instanceKlass 申请的内存空间大小实际上是 instanceKlass 类型本身所占的内存大小与 vtable、itable、static fields 及 oopMap 的大小之和，这种逻辑在代码中得到了体现，我们看 JDK 6 中 oopFactory::new_instanceKlass() 接口的实现逻辑：

清单：/src/share/vm/oops/instanceKlassKlass.cpp

功能：为 instanceKlass 申请内存空间

```

klassOop
instanceKlassKlass::allocate_instance_klass(int vtable_len, int itable_len,
                                              int static_field_size,
                                              unsigned nonstatic_oop_map_count,
                                              ReferenceType rt, TRAPS) {

```

```

const int nonstatic_oop_map_size =
    instanceKlass::nonstatic_oop_map_size(nonstatic_oop_map_count);
int size = instanceKlass::object_size(align_object_offset(vtable_len) +
align_object_offset(itable_len) + static_field_size + nonstatic_oop_map_size);

// Allocation
KlassHandle h_this_klass(THREAD, as_klassOop());
KlassHandle k;
if (rt == REF_NONE) {
    instanceKlass o;
    k = base_create_klass(h_this_klass, size, o.vtbl_value(), CHECK_NULL);
} else {
    instanceRefKlass o;
    k = base_create_klass(h_this_klass, size, o.vtbl_value(), CHECK_NULL);
}
// ...
}

```

与为常量池和方法对象申请内存空间的实现逻辑一样,在为 instanceKlass 申请内存空间时,oopFactory 也是先计算所要申请的内存大小,然后调用相应的接口进行申请。在这段代码中可以看到,所要申请的预留内存空间大小 size 的计算逻辑是 vtable、itable、static fields 及 oopmap 之和,因此最终静态字段一定在这个预留空间中。

在 JDK 6 中,JVM 在 instanceKlass 的内存空间末尾预留出足够的空间,存放虚方法表 vtable、接口表 itable、静态字段信息表及 Java 类中的引用类型表 oopMap,这 4 张表存放的顺序依次是 vtable、itable、static fields 和 oopMap,这是由 JVM 的源码所规定的,源码如下:

清单: /src/share/vm/oops/instanceKlass.hpp

功能: vtable、itable 和 oopMap 的存放顺序

```

// 获取 instanceKlass 对象实例在内存中所占用的空间大小
static int header_size() {
    return align_object_offset(oopDesc::header_size() +
sizeof(instanceKlass)/HeapWordSize);
}

// 获取 vtable 的起始偏移量,该偏移量正是 instanceKlass 类型本身的大小
static int vtable_start_offset() {
    return header_size();
}

// 获取 vtable 的内存地址
// 该地址值为 instanceKlass 实例对象的内存首地址 + instanceKlass 对象实例的内存大小
intptr_t* start_of_vtable() const {
    return ((intptr_t*)as_klassOop()) + vtable_start_offset();
}

```

```

// 获取 itable 的内存地址
// 该地址为 vtable 的内存首地址 + vtable 的长度, 该长度即为 itable 相对于 vtable 的偏移量
intptr_t* start_of_itable() const {
    return start_of_vtable() + align_object_offset(vtable_length());
}

// 静态字段的起始偏移量
HeapWord* start_of_static_fields() const {
    return (HeapWord*)(start_of_itable() + align_object_offset(itable_
length()));
}

// 获取 oopMap 的内存地址
// 该地址为 itable 的内存首地址 + itable 的长度, 该长度即为 oopMap 相对于 itable 的偏移量
OopMapBlock* start_of_nonstatic_oop_maps() const {
    return (OopMapBlock*)(start_of_itable() +
align_object_offset(itable_length()));
}

```

在 `instanceKlass.hpp` 文件中定义了 3 种接口, 分别用于获取 `vtable`、`itable` 和 `oopMap` 这 3 种数据在内存中的地址, 通过上面这段逻辑分析可知, 这 3 部分数据的存储顺序依次是 `vtable`、`itable` 和 `oopMap`。由此可知, JVM 调用 `oopFactory::new_instanceKlass()` 接口所创建的数据结构实际如图 10.2 所示。

图 10.2 中的这个内存数据结构, 便是 Java 类加载的最终产物, 也是 Java 类在内存中的对等体。JVM 根据这个数据结构, 能够获取 Java 类中所定义的一切元素。仔细观察图 10.2, 会发现其中有一项数据是 `c++ vtbl pointer`, 这是何物? 其实顾名思义, 这便是 C++ 类型对象中的虚方法表指针。由于 HotSpot 内部的 `klass` 都继承自基类 `Klass`, 而 `Klass` 类又继承自 `Klass_vtbl`, 但是 `Klass_vtbl` 中却有一个虚方法 `unused_initial_virtual()`。前面在讲解 Java 的虚方法表 `vtable` 的实现机制时曾经讲过, 如果 C++ 类中也包含虚方法, 则编译器会在 C++ 实例对象头部插入虚方法表的指针, 其道理与 Java 的虚方法表一样, 都是为了实现多态。不过在 JDK 6 里面让所有的 `klass` 类型都继承自 `Klass_vtbl`, 是为了更好地管理 `vtable`, 方便方法区的内存回收, 但是到了 JDK 8 就没有这玩意儿了, 因为所有的 `klass` 都继承了 `Metadata` 类。

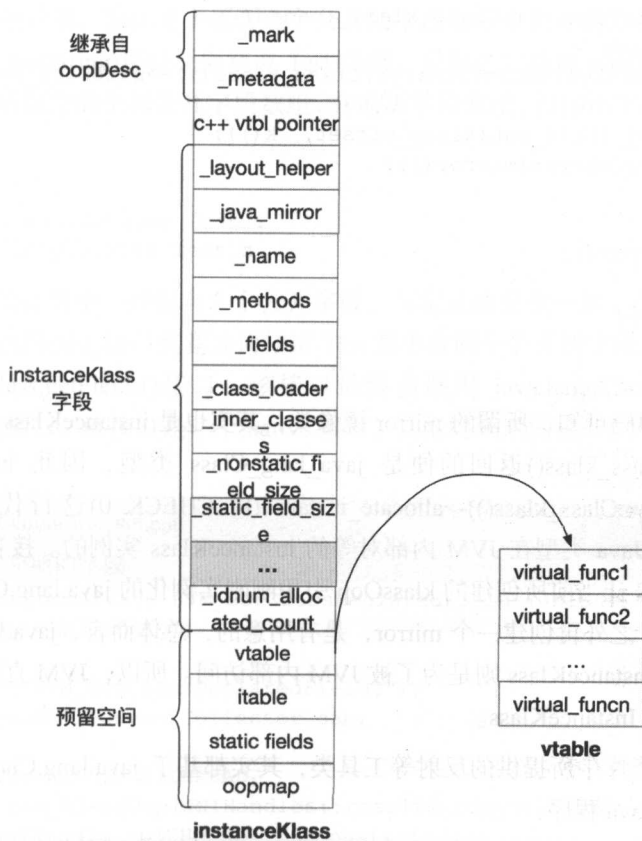


图 10.2 instanceKlass 的内存布局结构图

10.2.1 类加载——镜像类与静态字段

类加载的最终结果便是在 JVM 的方法区创建一个与 Java 类对等的 `instanceKlass` 实例对象，但是在 JVM 创建完 `instanceKlass` 之后，又创建了与之对等的另一个镜像类——`java.lang.Class`。在 JDK 6 中，创建镜像类的逻辑被包含在 `instanceKlassKlass::allocate_instance_klass()` 函数中，在该函数的末尾执行 `java_lang_Class::create_mirror()` 调用，该接口实现逻辑如下：

清单：/src/share/vm/classfile/javaClasses

功能：创建镜像类

```
oop java_lang_Class::create_mirror(KlassHandle k, TRAPS) {
    int computed_modifiers = k->compute_modifier_flags(CHECK_0);
    k->set_modifier_flags(computed_modifiers);
```



```

    if (SystemDictionary::Class_klass_loaded()) {
        Handle mirror =
instanceKlass::cast(SystemDictionary::Class_klass())->allocate_permanent_instance(CHECK_0);
        mirror->obj_field_put(klass_offset, k());
        k->set_java_mirror(mirror());

        // ...
        return mirror();
    } else {
        return NULL;
    }
}

```

通过观察这段源码可知,所谓的 mirror 镜像类,其实也是 instanceKlass 的一个实例对象, SystemDictionary::Class_klass() 返回的便是 java_lang_Class 类型,因此 instanceMirrorKlass::cast(SystemDictionary::Class_klass())->allocate_instance(k, CHECK_0) 这行代码就是用来创建 java.lang.Class 这个 Java 类型在 JVM 内部对等的 instanceKlass 实例的。接着通过 k->set_java_mirror(mirror()) 调用,让当前所创建的 klassOop 引用刚刚实例化的 java.lang.Class 对象。JVM 之所以在 instanceKlass 之外再创建一个 mirror,是有用意的,总体而言,java.lang.Class 是为了被 Java 程序调用,而 instanceKlass 则是为了被 JVM 内部访问。所以,JVM 直接暴露给 Java 的是 java_mirror,而不是 InstanceKlass。

事实上,JDK 类库中所提供的反射等工具类,其实都基于 java.lang.Class 这个内部镜像实现。例如下面这个 Java 程序:

清单: /Test.java

功能: 演示 Java 的反射功能

```

public class Test {
    public Integer i;
    public String s;

    public int add(int a, int b){
        return a + b;
    }

    public static void main(String[] args) throws Exception {
        Class klass = Class.forName("Test");
        System.out.println(klass.getFields().length);
        for(int i = 0; i < klass.getFields().length; i++){
            System.out.println(klass.getFields()[i]);
        }
    }
}

```

该示例 Java 类很简单, Test 类中包含 2 个公开的字段和一个公开的方法, 在 main()方法中通过 java.lang.Class.forName(String)接口反射获取 Test 类型, 反射之后通过 java.lang.Class.getFields()接口获取 Test 类中所包含的全部公开字段数组, 并遍历字段数组, 打印出字段名。运行该程序, 输出如下:

```
2
public java.lang.Integer Test.i
public java.lang.String Test.s
```

打印结果显示 Test 类中一共包含 2 个公开字段, 与定义的完全一致。在这里, 重点研究的是, java.lang.Class.getFields()接口究竟如何知道 Test 类中有两个公开的字段。源码面前无秘密。首先看 java.lang.Class.getFields()接口, 该接口最终会调用 java.lang.Class.getDeclaredFields0 (boolean publicOnly)接口, 该接口是一个 native 接口, 其最终调用的接口位于 HotSpot 内部的函数中, 该函数如下:

清单: /src/share/vm/prims/jvm.cpp

功能: 获取类中声明的字段

```
JVM_ENTRY(jobjectArray, JVM_GetClassDeclaredFields(JNIEnv *env, jclass
ofClass, jboolean publicOnly))
{
    JVMWrapper("JVM_GetClassDeclaredFields");
    JvmtiVMObjectAllocEventCollector oam;

    instanceKlassHandle k(THREAD,
java_lang_Class::as_klassOop(JNIHandles::resolve_non_null(ofClass)));
    constantPoolHandle cp(THREAD, k->constants());

    // 执行类的链接阶段
    k->link_class(CHECK_NULL);

    // 通过 k->fields() 获取类中的全部字段
    typeArrayHandle fields(THREAD, k->fields());
    int fields_len = fields->length();

    // ...

    return (jobjectArray) JNIHandles::make_local(env, result());
}
JVM_END
```

上面这个 JVM_GetClassDeclaredFields()函数便是 java.lang.Class.getDeclaredFields0 (boolean publicOnly)这个 Java 类方法所对应的内部实现。由于 java.lang.Class.getDeclaredFields0 (boolean publicOnly)方法是类的成员方法, 因此该方法包含一个隐藏的入参 this, this 指向 java.lang.Class

类型实例自己，所以调用的 `JVM_GetClassDeclaredFields()` 函数的第 2 个入参 `ofClass` 便是 `java.lang.Class` 类型实例。同时，在执行上面这个 `JVM_GetClassDeclaredFields()` 函数调用时，说明其前面的一个步骤——`Class klass = Class.forName("Test")` 已经执行完了，此时在 JVM 内部的 `klass` 实例，实际上是 `Test` 类型在 JVM 内部的镜像类，虽然 `java.lang.Class` 仅仅是一个镜像类，但是也保存了 `Test` 这个 Java 类中的全部信息，所以在 `JVM_GetClassDeclaredFields()` 函数中能够获取 `Test` 类中的全部字段。这便是 Java 反射的原理。通过本示例也可以知道，Java 的反射是离不开 `java.lang.Class` 这个镜像类的。

如果思维再放得开阔一点，可以这样认为，即使 JVM 内部没有安排 `java.lang.Class` 这么一个媒介作为面向对象反射的基础，那么 JVM 也必然要定义另外类，假设这个类就叫作 `Reflection`，这个类能够直接被 Java 程序开发者使用，那么 `Reflection` 这个类也必然需要在 JVM 内部与所要反射的目标 Java 类所对应的 `instanceKlass` 之间建立联系，能够让 Java 开发者通过这个 `Reflection` 类反射出目标 Java 类的字段、方法等全部信息。从这个意义上而言，`java.lang.Class` 并非是偶然有的，而是必然，是 Java 这种面向对象的语言与虚拟机实现机制这两种规范下的必然技术实现，如果非要说有巧合的话，那便是恰好叫了“`java.lang.Class`”这个类名。

既然 `java.lang.Class` 是一个必然的存在，所以每次 JVM 在内部为 Java 类创建一个对等的 `instanceKlass` 时，都要再创建一个对应的 `Class` 镜像类，作为反射的基础。

刚才讲过，在 JDK 6 中，静态字段会存储在 `instanceKlass` 的预留空间里，在 JVM 为 `instanceKlass` 申请内存空间时已经为静态字段预留了空间，而在创建完 `instanceKlass` 之后，JVM 在 `ClassFileParser::parseClassFile()` 函数中调用 `this_klass->do_local_static_fields(&initialize_static_field, CHECK_(nullHandle))` 对这部分内存空间进行初始化，`do_local_static_fields()` 函数的实现如下：

清单：/src/share/vm/classfile/classFileParser.cpp

功能：为 Java 类中静态字段分配空间

```
void instanceKlass::do_local_static_fields(void f(fieldDescriptor*, TRAPS),
TRAPS) {
    instanceKlassHandle h_this(THREAD, as_klassOop());
    do_local_static_fields_impl(h_this, f, CHECK);
}

void instanceKlass::do_local_static_fields_impl(instanceKlassHandle
this_oop, void f(fieldDescriptor* fd, TRAPS), TRAPS) {
    fieldDescriptor fd;
    int length = this_oop->fields()->length();
    for (int i = 0; i < length; i += next_offset) {
        fd.initialize(this_oop(), i);
        if (fd.is_static()) { f(&fd, CHECK); } // Do NOT remove {}! (CHECK macro
```

```
expands into several statements)
```

这段逻辑遍历 Java 类中的全部静态字段并逐个将其塞进 instanceKlass 的预留空间中。在这段逻辑中, 需要注意, instanceKlass::do_local_static_fields(void f(fieldDescriptor*, TRAPS), TRAPS) 函数的第一个入参是函数指针, 看上面这段逻辑, instanceKlass::do_local_static_fields(void f(fieldDescriptor*, TRAPS), TRAPS) 内部调用了 instanceKlass::do_local_static_fields_impl(instanceKlassHandle this_oop, void f(fieldDescriptor* fd, TRAPS), TRAPS), 而在后者内部则通过函数指针 f 调用其指向的函数。那么指针 f 指向哪个函数呢?

在 ClassFileParser::parseClassFile() 函数中调用 instanceKlass::do_local_static_fields(void f(fieldDescriptor*, TRAPS), TRAPS) 时, 所传入的函数指针是 &initialize_static_field, 所以该指针指向的函数如下:

清单: /src/share/vm/classfile/classFileParser.cpp

功能: 初始化静态字段

```
static void initialize_static_field(fieldDescriptor* fd, TRAPS) {
    KlassHandle h_k (THREAD, fd->field_holder());
    if (fd->has_initial_value()) {
        BasicType t = fd->field_type();
        switch (t) {
            case T_BYTE:
                h_k()->byte_field_put(fd->offset(), fd->int_initial_value());
                break;
            case T_BOOLEAN:
                h_k()->bool_field_put(fd->offset(), fd->int_initial_value());
                break;
            case // ...
        }
    }
}
```

在该函数中, 通过调用 h_k()->*_field_put() 系列接口, 将不同类型的静态字段存储到 instanceKlass 对象实例的预留内存空间中, 如此便完成了 Java 类中静态字段的存储。而在 JDK 8 中, 静态字段不再存储于 instanceKlass 预留空间, 而是转移到 instanceKlass 的镜像类——java.lang.Class 的预留空间里去, 因此在 JDK 8 的源码中, 上面的这个 initialize_static_field() 函数定义到 javaClasses.cpp 中了。同时, 创建 mirror 镜像类的接口也不再在 java_lang_Class::create_mirror() 函数中调用, 而是在 ClassFileParser::parseClassFile() 函数中调用。虽然调用的地方不同了, 但是函数实现的内部机制并没有从根本上发生变化, 因此从这一点上看, JDK 6 和 JDK 8 并没有做很大的变更。JDK 8 之所以要将静态字段从 instanceKlass 迁移到 mirror 中, 也不是没有道理,

毕竟静态字段并非 Java 类的成员变量，如果从数据结构这个角度看，静态字段不能算作 Java 类这个数据结构的一部分，因此 JDK 8 将静态字段转移到 mirror 中。从反射的角度看，静态字段放在 mirror 中是合理的，毕竟在进行反射时，需要给出 Java 类中所定义的全部字段，无论字段是不是静态类型。例如，将上面的 Test 类做个修改，在里面增加一个 static 类型的公开字段，则最终的打印结果会包含该字段。

综上所述，对于 JDK 6 而言，类加载阶段所产出的最终结果便是如图 10.3 所示的这两个实例对象。

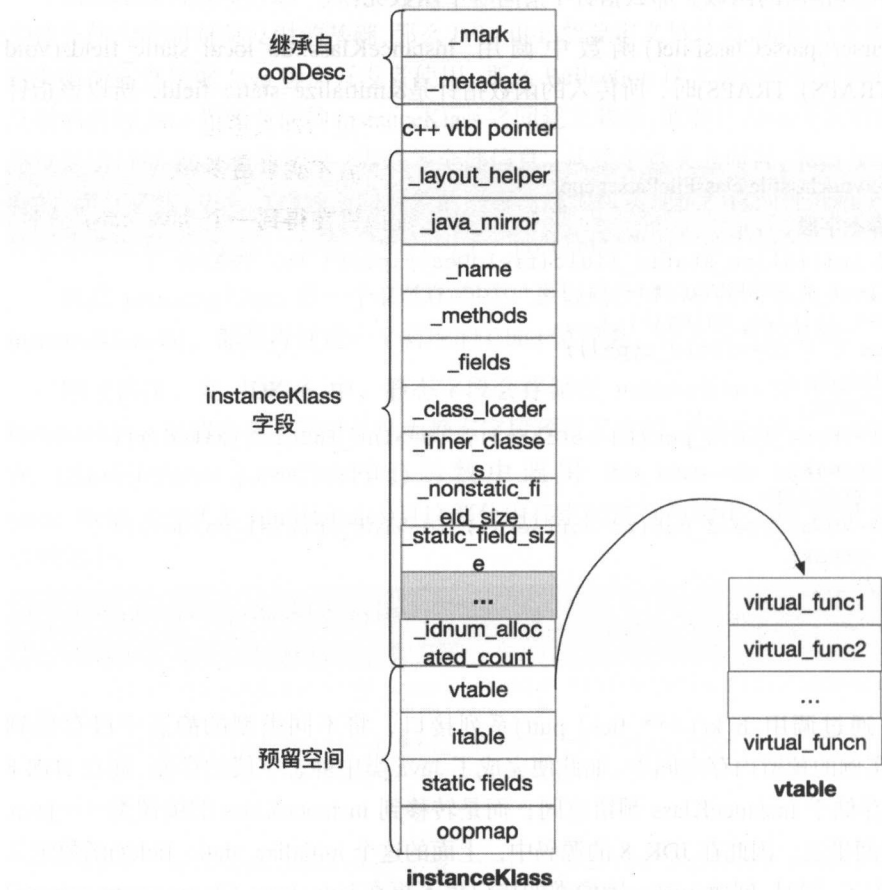


图 10.3 java 类加载阶段所产生的结果

在 JDK 6 中，由于 mirror 也是一个 `instanceKlass`，因此其包含了 `instanceKlass` 所包含的一切字段。

10.2.2 Java 主类加载机制

到上一节为止, Java 类加载的过程终于全部讲完了。在前面章节详细讲解了常量池解析、字段解析、方法解析、instanceKlass 创建及镜像类的创建。之所以要逐个详细讲解, 一方面是因为 JVM 使用 C/C++ 编写而成, 而 C/C++ 语言本身就比较 Java 语言更具难度, 相信只要不是直接从事 JVM 开发的道友, 阅读起来都会比较吃力, 里面有太多的内存分配、回收、指针、类型转换的内容, 笔者作为 Java 开发者, 阅读过程中也费了无数脑筋, 相当不轻松, 因此笔者感同身受, 将一些比较关键的源代码和算法详细描述出来, 这是自己辛苦阅读的一种沉淀, 相信也会帮助很多对 C/C++ 语言不够熟悉的道友。另一方面是因为 JVM 作为虚拟机, 里面涉及的计算机基础知识多而杂, 几乎覆盖了方方面面, 其实现也复杂, 然而其过程也精彩, 所以虽然阅读的过程痛苦, 但是结果却是快乐的, 理解了原理之后再次面对 Java 程序, 会有一种“一览众山小”之快感, 你就是 JVM 世界里的神, 做神的感觉, 其美妙不足为外人道也, 而这种享受也是支持笔者这两年里一直坚持写下去的最大动力。有苦有乐, 生活才能丰富多彩。

牛皮吹完, 我们应该总结一下类加载的整体过程了。虚拟机在得到一个 Java class 文件流之后, 接下来要完成的主要步骤如下:

- (1) 读取魔数与版本号。
- (2) 解析常量池, parse_constant_pool()。
- (3) 解析字段信息, parse_fields()。
- (4) 解析方法, parse_methods()。
- (5) 创建与 Java 类对等的内部对象 instanceKlass, new_instanceKlass()。
- (6) 创建 Java 镜像类, create_mirror()。

以上便是一个 Java 类加载的核心流程。了解了类加载的核心流程之后, 也许聪明的你会忍不住想, Java 类的加载到底何时才会被触发呢? Java 类加载的触发条件比较多, 其中比较特殊的便是 Java 程序中包含 main() 主函数的类——这种类一般也被称作 Java 程序的主类。Java 主类的加载由 JVM 自动触发——JVM 执行完自身的若干初始化逻辑之后, 第一个加载的便是 Java 程序的主类。总体上而言, Java 主类加载的链路如下:

```
java.c::JavaMain(): 执行 mainClass = LoadClass(env, classname)
java.c::LoadClass(): 执行 cls = (*env)->FindClass(env, buf)
jni.cpp::JNI_ENTRY(jclass, jni_FindClass(JNIEnv *env, const char *name)):
执行 loader = Handle(THREAD, SystemDictionary::java_system_loader())
jni.cpp::JNI_ENTRY(jclass, jni_FindClass(JNIEnv *env, const char *name)):
执行 result = find_class_from_class_loader(env, sym, true, loader,
protection_domain, true, thread) 加载主类
```



```

jvm.cpp::find_class_from_class_loader(): 执行 klassOop klass =
SystemDictionary::resolve_or_fail(name, loader, protection_domain, throwError !=
0, CHECK_NULL)
    SystemDictionary::resolve_or_fail()
        SystemDictionary::resolve_or_null()
            SystemDictionary::resolve_instance_class_or_null(): 执行 k =
load_instance_class(name, class_loader, THREAD) (Do actual loading)
                SystemDictionary::load_instance_class()
                    JavaCalls::call_virtual();
                        java.lang.ClassLoader.loadClass(String)
                            sun.misc.AppClassLoader.loadClass(String, boolean)
                                java.lang.ClassLoader.loadClass(String, boolean)
                                    java.net.URLClassLoader.findClass(final String)
                                        java.net.URLClassLoader.defineClass(String, Resource)
                                            java.lang.ClassLoader.defineClass(String,
java.io.ByteBuffer, ProtectionDomain)
                                                native java.lang.ClassLoader.defineClass0()
                                                    ClassLoader.c::Java_java_lang_ClassLoader_defineClass1()
                                                        jvm.cpp::JVM_DefineClassWithSource()
                                                            jvm.cpp::jvm_define_class_common()
                                                                SystemDictionary.cpp::resolve_from_stream()
                                                                    ClassFileParser.cpp::parseClassFile()

```

上面是 Java 程序 main 主类加载的整体链路，该调用链路的核心逻辑如下：

(1) JVM 启动后，操作系统会调用 `java.c::main()` 主函数，从而进入 JVM 的世界。`java.c::main()` 方法调用 `java.c::JavaMain()` 方法，`java.c::JavaMain()` 方法主要执行 JVM 的初始化逻辑，初始化完毕之后，便会搜索 Java 程序的 `main()` 主函数所在的类，也即“主类”，找到主类的类名之后，便会调用 `mainClass = LoadClass(env, classname)` 对主类进行加载。

(2) `LoadClass(env, classname)` 方法是 `java.c::LoadClass()` 方法，而后者执行 `cls = (*env)->FindClass(env, buf)` 来寻找主类。

(3) `(*env)->FindClass(env, buf)` 函数首先跳转到 `jni.cpp::JNI_ENTRY(jclass, jni_FindClass(JNIEnv *env, const char *name))`，`JNI_ENTRY` 是一个宏，在预编译阶段便已展开，这个宏作用的结果是：`(*env)->FindClass(env, buf)` 最终会调用 `jni.cpp::jni_FindClass(JNIEnv *env, const char *name)` 函数。

`jni.cpp::jni_FindClass(JNIEnv *env, const char *name)` 函数先调用 `loader = Handle(THREAD, SystemDictionary::java_system_loader())` 获取类加载器。Java 程序主类的类加载器默认是系统加载器，该加载器是 JDK 类库中定义的 `sun.misc.AppClassLoader`，关于该加载器的细节会在后文详述。JVM 体系中加载器的继承关系如图 10.4 所示。

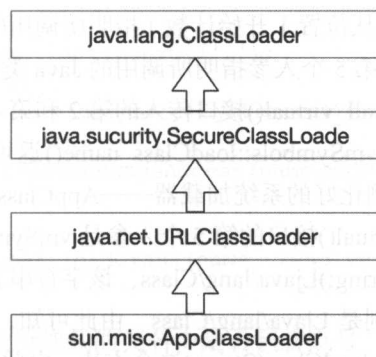


图 10.4 JVM 系统加载器的继承关系

由图 10.4 可知，系统加载器所继承的顶级父类是 `java.lang.ClassLoader`，这是 JDK 类库所提供的核心加载器。事实上，无论 Java 程序内部有没有自定义类加载器，最终都会调用 `java.lang.ClassLoader` 所提供的几个 native 接口完成类的加载，这些接口主要包括如下 3 种：

```

private native Class<?> defineClass0(String name, byte[] b, int off, int len,
    ProtectionDomain pd);
private native Class<?> defineClass1(String name, byte[] b, int off, int len,
    ProtectionDomain pd, String source);
private native Class<?> defineClass2(String name, java.nio.ByteBuffer b, int
    off, int len, ProtectionDomain pd, String source);
  
```

Java 主类的加载也无法绕过这 3 个接口。

`jni.cpp::JNI_ENTRY(jclass, jni_FindClass(JNIEnv *env, const char *name))` 函数内部获取到系统加载器之后，接着便开始调用 `find_class_from_class_loader()` 接口加载主类，而后者则调用 `SystemDictionary::resolve_or_fail()` 接口。

(4) `SystemDictionary::resolve_or_fail()` 接口经过一系列调用，最终调用 `SystemDictionary::resolve_instance_class_or_null()` 接口，该接口内部逻辑比较冗长，会经过层层判断，确认同一个加载器没有别的线程在加载同一个类，则最终会执行真正的加载，调用 `SystemDictionary::load_instance_class()` 接口，该接口内部执行如下调用：

```

JavaCalls::call_virtual(&result,
    class_loader,
    spec_klass,
    vmSymbols::loadClass_name(),
    vmSymbols::string_class_signature(),
    string,
    CHECK_(nh));
  
```

`JavaCalls::call_virtual()` 接口的主要功能是根据输入的参数，调用指定的 Java 类中的指定方

法。该接口的第 2 个人参（入参从位置 1 开始计数）指明所调用的 Java 类对应的 instance，第 4 个人参指明所调用的特定方法，第 5 个人参指明所调用的 Java 类的签名信息。当 JVM 执行 Java 程序主类加载时，向 `JavaCalls::call_virtual()` 接口传入的第 2 和第 4 个人参分别是 `class_loader` 和 `vmSymbols::loadClass_name()`，`vmSymbols::loadClass_name()` 返回的方法名是 `loadClass()`，而 `class_loader` 则是前置流程中实例化好的系统加载器——`AppClassLoader`，在 JVM 内部对等的实例对象。同时，`JavaCalls::call_virtual()` 接口的第 5 个人参是 `vmSymbols::string_class_signature()`，其返回的字符串是 `(Ljava/lang/String;)Ljava/lang/Class`，该字符串表示所调用的 Java 方法的入参是 `Ljava/lang/String`，而返回值则是 `Ljava/lang/Class`。由此可知，当 JVM 加载 Java 程序的主类时，最终会调用 `AppClassLoader.loadClass(String)` 这个方法。由此，JVM 的流程便转移到了 Java 的世界，进入到了 Java 类的逻辑流之中。

例 11-1 `JavaCalls::call_virtual()` 接口的第 6 个人参则包含所调用的 Java 方法所需的全部入参信息，在 JVM 加载 Java 应用程序主类时，向 `JavaCalls::call_virtual()` 接口所传入的第 6 个人参是 `string`，在 `SystemDictionary::load_instance_class()` 函数中，该入参封装了所需要加载的 Java 类的全限定名称，最终这个全限定名称将作为 `java.lang.AppClassLoader.loadClass(String)` 接口的入参，系统加载器据此加载目标 Java 类。

`JavaCalls::call_virtual()` 接口最终会调用 `JavaCalls::call()` 接口，`JavaCalls::call()` 接口调用 `JavaCalls::call_helper()`，而后者则会调用 `StubRoutines::call_stub()` 例程，对于该例程，阅读过全书的小伙伴一定不会陌生，该例程在本书前面专门花了一章去讲解，有不清楚的小伙伴可以回过去仔细阅读。总体而言，该例程在运行期对应着一段机器码，其作用是辅佐 JVM 执行 Java 类方法。这里不得不提一句，JVM 作为一款虚拟机，其本身由 C/C++ 语言写成，但是 JVM 是为执行 Java 字节码文件而生的，因此 JVM 内部必然有一套机制能够从 C/C++ 程序调用 Java 类中的方法，这套机制便通过 `JavaCalls` 类来实现，该类中定义了各种 `call_*()` 接口，这些接口最终都要调用 `StubRoutines::call_stub()` 例程，从而辅佐 JVM 执行 Java 方法。

事实上，`JavaCalls::call_virtual()` 接口在 JVM 内部是一个很常用的接口，大凡涉及 Java 类成员方法的调用，最终都会经过该接口。

（5）经过上一个步骤，JVM 最终会调用 `sun.misc.AppClassLoader.loadClass(String)` 接口加载 Java 应用程序的主类。`AppClassLoader` 继承自 `java.lang.ClassLoader` 这个基类，`java.lang.ClassLoader.loadClass(String)` 方法调用 `loadClass(String, boolean)` 方法，由于继承的关系，实际调用的是 `sun.misc.AppClassLoader.loadClass(String, boolean)` 方法，该方法的实现逻辑如下：

清单：/src/sun/misc/Launcher.java

功能：系统加载器加载类的逻辑

```
public Class loadClass(String name, boolean resolve) throws
```

```

ClassNotFoundException{
    int i = name.lastIndexOf('.');
    if (i != -1) {
        SecurityManager sm = System.getSecurityManager();
        if (sm != null) {
            sm.checkPackageAccess(name.substring(0, i));
        }
    }
    return (super.loadClass(name, resolve));
}
}

```

这段代码逻辑是，先判断所加载的类名中是否包含点号“.”，如果包含则说明传入的一定是类的全限定名，包含了包名，则 JVM 调用 SecurityManager 模块检查包的访问权限。通过访问权限验证之后，则调用 super.loadClass(name, resolve) 方法。由于继承关系，super.loadClass(name, resolve) 方法其实调用的是 java.lang.ClassLoader.loadClass(String name, boolean resolve) 方法，该方法的主要逻辑如下：

清单：/src/java/lang/ClassLoader

功能：java.lang.ClassLoader.loadClass(String name, boolean resolve) 方法逻辑

```

protected synchronized Class<?> loadClass(String name, boolean resolve) throws
ClassNotFoundException {
    Class c = this.findLoadedClass(name);
    if(c == null) {
        try {
            if(this.parent != null) {
                c = this.parent.loadClass(name, false);
            } else {
                c = this.findBootstrapClassOrNull(name);
            }
        } catch (ClassNotFoundException var5) { ; }

        if(c == null) {
            c = this.findClass(name);
        }
    }

    if(resolve) {
        this.resolveClass(c);
    }

    return c;
}

```

在 java.lang.ClassLoader.loadClass(String name, boolean resolve) 方法中，首先通过 findLoadedClass(name) 方法判断当前加载器是否加载过指定的类，如果没有加载，则判断当前加

载器的 parent 是否为 null，如果不为 null，则调用 `parent.loadClass(name, false)` 方法，通过父加载器加载指定的 Java 类。AppClassLoader 的父加载器是 ExtClassLoader，这是扩展类加载器，用于加载 JDK 中指定路径下的扩展类，这种加载器不会加载 Java 应用程序的主类，所以程序流会进入 `if(this.parent != null){}` 代码块，但是 `parent.loadClass(name, false)` 返回 null。接着 `java.lang.ClassLoader.loadClass(String name, boolean resolve)` 方法只能通过调用 `this.findClass(name)` 来加载 Java 主类。

`java.lang.ClassLoader.findClass(String)` 方法直接抛出异常，因此该类注定要由子类来实现。对于系统类加载器 AppClassLoader，其继承自 URLClassLoader，因此 `java.lang.ClassLoader.findClass(String)` 方法实际指向 `java.net.URLClassLoader.findClass(String)`。`java.net.URLClassLoader.findClass(String)` 方法最终调用 `java.lang.ClassLoader.defineClass1()` 这一 native 接口，这是一个本地接口，由本地类库实现。openjdk 项目包含了 JDK 核心 Java 类库中的全部本地实现，`java.lang.ClassLoader.defineClass1()` 所对应的本地实现是 `ClassLoader.c::Java_java_lang_ClassLoader_defineClass1()`，有兴趣的道友可自行查看下其实现，这里就不贴代码了，以免占用过多篇幅。通过调用 `java.lang.ClassLoader.defineClass1()` 接口，Java 程序流又转移到 JVM 内部，因此 Java 类的加载最终仍然是通过 JVM 本地类库得以实现。

`ClassLoader.c::Java_java_lang_ClassLoader_defineClass1()` 调用 `jvm.cpp::JVM_DefineClassWithSource()`，`jvm.cpp::JVM_DefineClassWithSource()` 调用 `jvm.cpp::jvm_define_class_common()`，而后者则调用 `SystemDictionary.cpp::resolve_from_stream()` 接口来加载 Java 主类。在 `SystemDictionary.cpp::resolve_from_stream()` 接口中，终于开始调用 `ClassFileParser.cpp::parseClassFile()` 这个函数来解析 Java 主类，并最终创建 Java 主类在 JVM 内部的对等体——`klassInstance`，由此完成 Java 主类的加载。

10.2.3 类加载器的加载机制

体现 Java 语言强大生命力和巨大魅力的关键因素之一便是，Java 开发者可以自定义类加载器来实现类库的动态加载，加载源可以是本地的 JAR 包，也可以是网络上的远程资源。通过类加载器可以实现非常绝妙的插件机制，这方面的实际应用案例举不胜数，例如，著名的 OSGI 组件框架，再如 Eclipse 的插件机制。类加载器为应用程序提供了一种动态增加新功能的机制，这种机制无须重新打包发布应用程序就能实现。同时，自定义加载器能够实现应用隔离，例如 Tomcat、Spring 等中间件和组件框架都在内部实现了自定义的加载器，并通过自定义加载器隔离不同的组件模块。这种机制比 C/C++ 程序要好太多，想不修改 C/C++ 程序就能为其新增功能，几乎是不可能的，仅仅一个兼容性便能阻挡住所有美好的设想。

Java 应用程序自定义类加载器很简单,只需要继承 `java.lang.ClassLoader` 类,然后覆盖它的 `findClass(String name)` 方法即可。使用自定义类加载器来加载 Java 类的使用方式通常如下:

```
MyClassLoader mcl = new MyClassLoader();
Class klass = mcl.loadClass("com.***.Test");
Test t = (Test)klass.newInstance();
```

通过这段示例程序可以看出,使用自定义类加载器进行 Java 类加载时,首先需要调用加载器的 `loadClass()` 接口完成 `java.lang.Class` 类的加载,然后才能实例化所要加载的类型实例。自定义类加载器的 `loadClass()` 方法所完成的类加载,便是 Java 类生命周期 7 个阶段(加载→验证→准备→解析→初始化→使用→卸载)中的类加载阶段。

有很多自定义的类加载器会重写 `java.lang.ClassLoader` 中的很多接口,实现起来非常复杂,但是无论多么复杂的自定义类加载器,最终都会调用 `java.lang.ClassLoader.defineClass*()` 系列本地接口,最终仍然会由 JVM 内部的本地实现来完成实际的加载工作,在 JVM 内部创建与所要加载的目标 Java 类对等的 `klassInstance` 对象实例,从而完成 Java 类型的加载。

关于 `java.lang.ClassLoader.defineClass*()` 的本地接口实现机制,在前面讲述 Java 应用程序主类加载时已经讲解过,该接口最终会调用 `ClassFileParser.cpp::parseClassFile()` 这个 JVM 内部的函数完成 Java 类的解析,并创建内部对应的对象实例,将 Java 类从字节码文件格式完全转换成内存格式。

10.2.4 反射加载机制

除了通过自定义类加载器完成类的加载,也可以通过 `java.lang.Class.forName(String)` 反射接口完成类加载,不过该接口所完成的加载,包含了 Java 类生命周期 7 个阶段的前面 5 个——加载、验证、准备、解析和初始化。

`java.lang.Class.forName(String)` 最终会调用 `private static native Class<?> forName0()` 这个本地接口完成类加载。`openjdk` 提供了该方法的本地接口实现,如下(仅摘录核心逻辑):

清单: `/src/java/lang/Class.c`

功能: 通过反射加载类

```
JNIEXPORT jclass JNICALL
Java_java_lang_Class_forName0(JNIEnv *env, jclass this, jstring classname,
                               jboolean initialize, jobject loader)
{
    char *cname;
    jclass cls = 0;
    // ...
```



```

    cls = JVM_FindClassFromClassLoader(env, cname, initialize,
                                       loader, JNI_FALSE);
    return cls;
}

```

java.lang.Class.forName()的本地接口调用 jvm.cpp::JVM_FindClassFromClassLoader()进行类加载，而后者会调用 jvm.cpp::find_class_from_class_loader()接口，前面讲述 Java 主类加载时绘制了 Java 主类的加载链路，Java 主类加载也会经过 jvm.cpp::find_class_from_class_loader()接口，因此两者的后续流程都相同，最终仍然会从 JVM 内部发起对 java.lang.ClassLoader.loadClass(String)接口的调用，从而完成类的真正加载。由此可知，在使用反射机制加载类时，最终仍然走了 java.lang.ClassLoader.loadClass(String)这个方法。其中的技术实现细节前面都已详细阐述过，这里不再赘述。

10.2.5 import 与 new 指令

在硬编码阶段，如果要想使用某个类，必须先 import 进来。但是纵观 Java 的字节码指令，以及编译后的 Java class 字节码文件，里面其实并没有任何关于 import 关键字的解释或痕迹，到了编译阶段，import 关键字就这样悄无声息地消失了。

事实上，import 语句仅仅是个语法糖，是为了不写那一长串的全限定名。import 关键字并没有任何关联的运行时行为，更不会导致类的加载。它的存在纯粹是为了方便写代码，让大家可以把别的 package 的“名字”引入到当前源码文件里直接用。否则，每次在 new 一个 Java 类时，都必须在类名前面补全完整的 package 包名。而 Java 类的加载，则使用了“延迟加载”机制，仅在第一次被使用时才会发生真正的加载，而与是否使用了 import 关键字无关。延迟加载的机制在后文会详细讲解。

虽然 import 语句没有对应的指令，也不会导致类的加载，但是所 import 进来的包名则会被写入 Java class 文件的常量池中，作为字符串储存起来，以用于类加载时的验证和解析。

至于 import 进来的类究竟啥时候会被加载，有好几种情况。例如，使用 new 关键字，或者读写类的静态变量，或者通过反射加载类。本节便来看看使用 new 关键字对类进行实例化的时候，JVM 内部是如何完成类的加载的。

在 32 位 Intel 处理器上，new 指令对应的实现在 templateTable_x86_32.cpp::TemplateTable::_new()函数中。使用 new 指令加载类时，如果一个类尚未被加载或者未被链接过，则会进入慢分配流程（后文会讲解类的分配机制），而慢分配主要通过调用 InterpreterRuntime::_new()函数完成。InterpreterRuntime::_new()函数会调用 constantPool->klass_at()接口来获取 new 指令后面所对应的 Java 类模板，该接口最终调用 constantPoolOopDesc::klass_at_impl()函数，后

者实现如下（下面仅摘录主要逻辑）：

清单：/src/share/vm/oops/constantPoolOop.cpp

功能：new 指令加载类

```

classOop constantPoolOopDesc::klass_at_impl(constantPoolHandle this_oop, int
which, TRAPS) {
    // .....
    { ObjectLocker ol(this_oop, THREAD);
        // 获取类加载器
        loader = Handle(THREAD, instanceKlass::cast(this_oop->pool_holder())->
class_loader());
    } // unlocking constantPool

    // ...
    if (do_resolve) {
        // 执行类加载
        klassOop k_oop = SystemDictionary::resolve_or_fail(name, loader, h_prot,
true, THREAD);
    }

    return (klassOop)entry.get_oop();
}

```

如果在应用程序中第一次使用某个类，且此时类尚未被加载进 JVM 内部，会走上上面这条链路，先获取类加载器，最终调用 `SystemDictionary::resolve_or_fail()` 接口进行类加载。而 Java 主类的加载也会走到这个接口中，因此后续链路与 Java 主类加载的链路完全相同，最终仍然会调用 `java.lang.ClassLoader.loadClass(String)` 这个 java 接口去执行类加载。本章前面已经详细讲解过 Java 主类的加载过程，因此这里不再赘述。由此可知，在第一次对某个 Java 类使用 `new` 关键字创建其实例对象时，如果类尚未加载过，则会进入上述流程先完成加载，再进行实例化。

10.3 类的初始化

完成类的加载后，经过链接，便会进入类的初始化阶段。所谓初始化，其实说白了就是调用 java 类的 `<clinit>()` 方法。前文已经讲过，该方法是编译器在编译期间自动生成的，当 Java 类中出现静态字段或者包含 `static {}` 块逻辑时，所编译出来的 Java 字节码文件中便会自动包含一个名为 `<clinit>` 的方法。该方法不能由程序员在 Java 程序中调用，只能由 JVM 在运行期调用，这个调用的过程便是 Java 类的初始化。注意，`<clinit>()` 方法并非类的构造函数。

JVM 规范规定，当遇到 `new`、`getstatic`、`invokestatic` 等字节码指令或者加载 Java 应用程序

主类或者其他一些情况时, 会执行类的初始化逻辑。下面以 new 指令为例, 说明 JVM 内部是如何一步一步调用<clinit>()方法的。

当使用 new 关键字来实例化一个 Java 类时, 如果该 Java 类是第一次被使用, 则必定会先执行加载→链接→初始化逻辑, 然后才能创建类实例对象。使用 new 关键字时, 在 32 位 Intel 处理器上, new 指令对应的实现在 templateTable_x86_32.cpp::TemplateTable::_new()函数中。使用 new 指令加载类时, 如果一个类尚未被加载和解析过, 则会进入慢分配流程, 慢分配流程调用 InterpreterRuntime::_new() 函数, 而 InterpreterRuntime::_new() 函数会调用 klass->initialize(CHECK)接口, 该接口的实现在 instanceClass.cpp 中, 该接口内部调用 instanceClass::initialize_impl(), 后者调用 instanceClass::call_class_initializer(), instanceClass::call_class_initializer()调用 instanceClass::call_class_initializer_impl()接口。该接口实现逻辑如下:

清单: /src/share/vm/oops/instanceClass.cpp

功能: Java 类的初始化逻辑

```
void instanceClass::call_class_initializer_impl(instanceClassHandle
this_oop, TRAPS) {
    // 获取 Java 类的<clinit>() 函数所对应的 method 对象
    methodHandle h_method(THREAD, this_oop->class_initializer());

    // 如果 Java 类没有<clinit>() 函数, 则不执行初始化逻辑, 跳过
    if (h_method() != NULL) {
        JavaCallArguments args; // No arguments
        JavaValue result(T_VOID);
        JavaCalls::call(&result, h_method, &args, CHECK); // 调用<clinit>() 函数
    }
}
```

这段逻辑其实比较简单, 先获取 Java 类的<clinit>()函数所对应的 method 对象, 接着通过 JavaCalls::call()接口执行初始化方法。如此便完成类的初始化。当然, 如果 Java 类中没有定义任何静态字段, 也没有 static{}逻辑块, 则编译后的字节码文件中自然不会包含<clinit>()方法, 则上面这段逻辑不会执行, 直接跳过。

前文讲过, 每一个类都有一个加载器, 并且不同加载器加载的同一个类无法相互转换。换言之, 如果先后使用 2 个不同的类加载器去加载同一个类, 则该类必定会先后被加载两次。同理, 该类的初始化逻辑也会先后被执行两次。看下面这段示例:

清单: /Test.java

功能: 使用不同的类加载器加载同一个类

```
public class Test {
    static {
        System.out.println("static logic. classLoader=" + Test.class.
```

```

getClassLoader());
    }

    public static void main(String[] args) throws Exception {
        ClassLoader loader = new ClassLoader() {
            @Override
            public Class<?> loadClass(String name) throws ClassNotFoundException {
                try {
                    String className = name.substring(name.lastIndexOf(".") + 1) + ".class";
                    InputStream is = getClass().getResourceAsStream(className);
                    if (is == null) {
                        return super.loadClass(name);
                    }
                    byte[] buffer = new byte[is.available()];
                    is.read(buffer);
                    return defineClass(name, buffer, 0, buffer.length);
                } catch (Exception e) {
                    throw new ClassNotFoundException(name);
                }
            }
        };

        System.out.println("start to load Test with custom classLoader");
        Object test = loader.loadClass("Test").newInstance();
        System.out.println("loaded Test with custom classLoader");

        Test t = new Test();
    }
}

```

在本示例程序中，为 Test 测试类加入了 static{} 逻辑块，并在里面打印一行文字。这样如果调用类的初始化逻辑，则可以通过打印结果观察到。在本示例程序的主函数中，自定义了一个类加载器，并使用该类加载器加载 Test 测试类。运行该程序，打印结果如下：

```

static logic. classLoader=sun.misc.Launcher$AppClassLoader@2d3fcd9d
start to load Test with custom classLoader
static logic. classLoader=Test$1@36f6e879
loaded Test with custom classLoader

```

通过打印结果可以观察到，在自定义的类加载器准备加载测试类之前，Test 类的 static{} 逻辑便被执行过一次。这是因为 Test 类包含主函数，因此其属于测试程序的主类，在 JVM 启动完成之后便会首先加载该类。使用自定义的类加载器再次加载 Test 类时，由于虽然测试类已经被加载过，但是由于这一次使用的类加载器发生了变化，因此 JVM 便又加载了它一次，因此 Test 的 static{} 块逻辑再次被调用。

10.4 类加载器

要想在 JVM 内部创建一个与 Java 类完全对等的结构模型，必须经过类加载器。类加载器的好处自不必多言，本节开始一起探讨类加载器的本质，类加载器到底是啥，与 JVM 内部究竟有哪些联系，所谓的双亲委派到底是怎么回事，JDK 的核心类库究竟是啥时候加载的……。

10.4.1 类加载器的定义

Java 体系中定义了 3 种类加载器，分别如下：

- ◎ Bootstrap ClassLoader(引导类加载器,缩写为 BCL)。加载指定的 JDK 核心类库，无法由 Java 应用程序直接引用。负责加载下述 3 种情况下所指定的核心类库：
 - %JAVA_HOME%/jre/lib 目录
 - -Xbootclasspath 参数所指定的目录
 - 系统属性 sun.boot.class.path 指定的目录中特定名称的 jar 包
- ◎ Extension ClassLoader(扩展类加载器,缩写为 ECL)。加载扩展类，拓展 JVM 的类库。该加载器加载下述两种情况下所指定的类库：
 - %JAVA_HOME%/jre/lib/ext 目录
 - 系统属性 java.ext.dirs 所指定的目录中的所有类库
- ◎ System ClassLoader(系统类加载器,缩写为 SCL)。加载 Java 应用程序类库，加载类库的路径由系统环境变量 ClassPath、-cp 或 系统属性 java.class.path 指定。

除了这 3 种加载器之外，Java 还能支持开发者自定义加载器，自定义的加载器大大丰富了 Java 中间件，在若干 Java 框架和组件中得到极其广泛的应用。通过下面这个测试程序，可以获取运行时 JVM 的各类加载器所加载的类路径：

清单：/Test.java

功能：获取 JVM 各种类加载器的类路径

```
public class Test {
    public static void main(String[] args) {
        System.out.println("引导类加载器加载路径: " + System.getProperty(
            "sun.boot.class.path"));
        System.out.println("扩展类加载器加载路径: " + System.getProperty(
            "java.ext.dirs"));
        System.out.println("系统类加载器加载路径" + System.getProperty(
            "java.class.path"));
    }
}
```

```

    }
}

```

在下文讲述系统类加载器和扩展类加载器的初始化机制时，会看到 JDK 核心类库是通过直接读取这几个系统属性来获取对应加载器的路径的。

从使用的角度看，虽然 JVM 提供了多种多样的类加载器，并且开发者可以自定义若干加载器，但是站在程序的角度看，其实 Java 体系一共只定义了两种类加载器，一种使用 C++ 语言定义，另一种则使用 Java 语言定义。使用 C++ 语言定义的种类加载器如下：

清单：/src/share/vm/classfile/classLoader.hpp

功能：使用 C++ 定义的种类加载器

```

class ClassLoader: AllStatic {
private:
    friend class LazyClassPathEntry;
    static ClassPathEntry* _first_entry;
    static ClassPathEntry* _last_entry;
    static PackageHashtable* _package_hash_table;
    static const char* _shared_archive;

    // 根据 Java 类名加载 Java 类
    static instanceKlassHandle load_classfile(Symbol* h_name, TRAPS);

    // 设置加载器所加载的类路径
    static void setup_bootstrap_search_path();

    // ...
public:
    // 初始化类加载器
    static void initialize();

    // ...
};

```

JVM 内部使用 C++ 定义的 ClassLoader，其实这便是传说中的 bootstrap class loader，即引导类加载器。该加载器内部所有的字段和函数都使用 static 修饰，因此该加载器并不需要实例化，当需要加载 Java 类时，直接调用静态函数。该加载器提供 setup_bootstrap_search_path() 接口用于设置加载器所要搜索的类路径，同时提供了一个最重要的方法——load_classfile()，来加载指定的 Java 类。该接口的实现如下：

清单：/src/share/vm/classfile/classLoader.cpp

功能：引导类加载器加载逻辑

```
instanceKlassHandle ClassLoader::load_classfile(Symbol* h_name, TRAPS) {
    // ...

    instanceKlassHandle h(THREAD, klassOop(NULL));

    ClassFileParser parser(stream);
    Handle class_loader;
    Handle protection_domain;
    TempNewSymbol parsed_name = NULL;
    instanceKlassHandle result = parser.parseClassFile(h_name,
                                                         class_loader,
                                                         protection_domain,
                                                         parsed_name,
                                                         false,
                                                         CHECK_(h));

    // add to package table
    if (add_package(name, classpath_index, THREAD)) {
        h = result;
    }
}

return h;
}
```

如果对前文讲解的 Java 字节码文件的常量池解析比较熟悉，则你对这段代码也不会感到陌生，没错，这个 C++ 加载器的加载接口直接调用了 `ClassFileParser::parseClassFile()` 接口来解析并加载 Java 类，并最终在 JVM 内部创建一个与 Java 类完全对等的 C++ 对象实例，完成类的加载。如果对 `ClassFileParser::parseClassFile()` 接口不够熟悉，建议翻看前面的章节，前文对该接口的讲解很详细。JVM 内部所定义的引导类加载器的实现十分干脆纯净，不像使用 Java 所定义类加载器那么“九弯十八绕”。不过，并不是所有的 Java 虚拟机都会使用 C++ 专门定义一个类加载器，有些 JVM 也使用 Java 语言来定义引导类加载器，只不过其实现仍然要依赖于 JVM 内部所提供的本地接口。是否使用 C++ 来定义引导类加载器并不重要，重要的是，无论是用 Java 编写的加载器，还是用其他语言编写的加载器，其最终目的都是要在 JVM 内部创建一个与 Java 类完全对等的结构体。只要能够实现这个目标，技术上怎么玩都是可以的。

了解了 C++ 定义类加载器，再看 Java 定义的加载器。使用 Java 语言定义类加载器，便是 JDK 核心类库中的 `java.lang.ClassLoader` 类。这里就不贴出该类的主要逻辑了，打开 IDE 便能查看。在 HotSpot 中，除引导类加载器 BCL 外，其余所有的类加载器——无论是 Java 体系所

提供的，还是开发者自定义的，都继承自 `java.lang.ClassLoader`，扩展类加载器与系统类加载器也不例外。扩展类加载器与系统类加载器都定义在 `sun.misc.Launcher` 类中，类名分别是 `ExtClassLoader` 和 `AppClassLoader`，这两个类加载器都继承自 `URLClassLoader`，而 `URLClassLoader` 则继承自 `java.lang.ClassLoader`。事实上，`java.lang.ClassLoader` 提供了绝大多数类加载功能，同时提供了最重要的 `define*` 系列的 `native` 接口，扩展类加载器与系统类加载器最终也是依靠 `java.lang.ClassLoader` 的本地接口方能完成 Java 类的加载。所以，可以这么说，扩展类加载器与系统类加载器仅仅是张皮而已，`java.lang.ClassLoader` 才是真正的“幕后主事”。

事实上，从 JDK 研发者的角度看，最初的 JDK 根本就没有所谓的类加载器这个概念，JDK 的核心类库直接通过调用 `ClassFileParser::parseClassFile()` 接口完成加载，而对于 Java 应用程序中的类库，则绕个弯子，通过调用 `native` 接口从而间接调用 `ClassFileParser::parseClassFile()` 接口完成加载。只不过为了 Java applet 而专门开发了类加载器。只可惜 Java applet 没有生根发芽，但是类加载器倒是遍地开花，生活得很好。看来技术也有“听天由命”的一面，有心栽花花不开，无心插柳柳成荫。

Java 体系所定义的 3 种类加载器——引导类加载器、扩展类加载器和系统类加载器，与开发者自定义的类加载器，它们之间存在一定的关系，这种关系如图 10.5 所示。

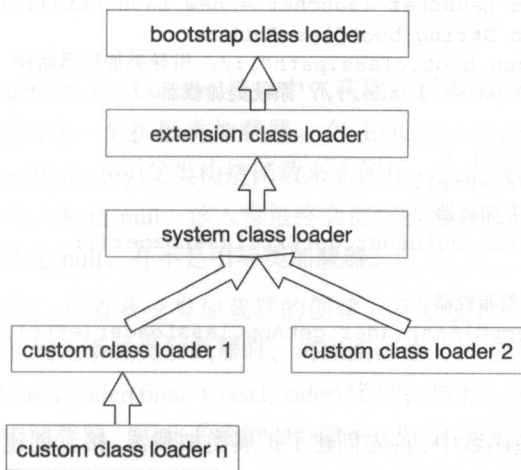


图 10.5 Java 类加载器之间的委派关系

图 10.5 所示的各种类加载器之间的关系，并非 Java 类面向对象的三大特性之一的那种“继承”关系，毕竟引导类加载器是使用 C++ 语言编写而成的，Java 类编写的类加载器想去继承也无法继承。图 10.5 所表达的联系，仅仅是类的委托加载机制，尤其是其中著名的“双亲委派”机制，为世人所乐道，这种机制将在下文描述。虽然这几种类加载器之间并没有直接的继承关

系,但是扩展类加载器、系统类加载器及用户自定义的加载器,却都继承自 `java.lang.ClassLoader` 这个基类。前文讲解过该类,当 JVM 加载 Java 主类时,最终也是通过 `java.lang.ClassLoader.loadClass(String)` 这一接口完成的。

图 10.5 所表达的委托加载关系,本质上通过 `java.lang.ClassLoader.parent` 字段实现,该字段表示父加载器。对于引导类加载器,并没有所谓的父加载器的概念,因为引导类加载器本身是随 JVM 的启动而初始化,并且该类中的字段和方法全部是静态字段和方法,并不需要实例化便能使用。而除了引导类加载器之外的所有类加载器,由于都继承自 `java.lang.ClassLoader` 这个基类,因此便都拥有 `parent` 字段,由于该字段被 `final private` 修饰,因此子类只能通过调用 `java.lang.ClassLoader` 的构造函数才能初始化该字段。对于扩展类加载器和系统类加载器,在 JVM 第一次加载 Java 类时会被创建,并完成其父加载器的设定。扩展类加载器与系统类加载器在 `sun.misc.Launcher` 类的构造函数中完成初始化,`sun.misc.Launcher` 类的构造函数逻辑如下(仅摘录主要逻辑):

清单: `/src/sun/misc/Launcher.java`

功能: 构造函数

```
public class Launcher {
    private static URLStreamHandlerFactory factory = new Factory();
    private static Launcher launcher = new Launcher(); // 启动器
    private static String bootClassPath =
        System.getProperty("sun.boot.class.path"); // 引导类加载器路径
    private ClassLoader loader; // 系统类加载器

    public Launcher() {
        ClassLoader extcl;
        // 创建扩展类加载器
        extcl = ExtClassLoader.getExtClassLoader();

        // 创建系统类加载器
        loader = AppClassLoader.getAppClassLoader(extcl);
    }
}
```

在 `Launcher` 类的构造函数中,首先创建了扩展类加载器,接着创建了系统类加载器。注意,扩展类加载器是在构造函数中定义的局部变量 `extcl`,而系统类加载器则是 `Launcher` 类的成员变量 `loader`。为何扩展类加载器不是一个类成员变量,而是一个局部变量呢?当构造函数执行完毕,这个扩展类实例变量不就被销毁了吗,那么 JVM 还如何使用该扩展类去加载扩展包呢?其实,当 `Launcher` 类的构造函数执行完之后,扩展类加载器实例对象并不会被 GC 回收,因为在创建系统类加载器的时候,扩展类加载器被设置为系统类加载器的 `parent`,因此当 JVM 想加载扩展类时,总是能够通过系统类加载器的 `parent` 属性获取到扩展类加载器,从而使用扩展类加

载器去加载相应的类库。

在 Launcher 类的构造函数中,调用 ExtClassLoader.getExternalClassLoader()接口创建扩展类加载器,该接口实现如下(仅摘录主要逻辑):

清单: /src/sun/misc/Launcher.java

功能: 创建扩展类加载器

```
static class ExtClassLoader extends URLClassLoader {
    // 创建扩展类加载器
    public static ExtClassLoader getExtClassLoader() throws IOException
    {
        final File[] dirs = getExtDirs();// 获取扩展类加载路径

        return new ExtClassLoader(dirs);// 实例化扩展类加载器
    }

    // 获取扩展类加载文件
    private static File[] getExtDirs() {
        String s = System.getProperty("java.ext.dirs");
        File[] dirs;
        // ...
        return dirs;
    }
}
```

在 ExtClassLoader.getExternalClassLoader()接口中先获取扩展类加载路径,最后直接通过 new ExtClassLoader(dirs)来实例化一个扩展类加载器。在 ExtClassLoader(File[])构造函数中,调用 super(getExtURLs(dirs), null, factory)父类构造函数来实例化一个加载器,注意,在调用父类构造函数时,所传入的第 2 个入参是 null,该入参最终会被赋值给 parent 属性。由此可见,扩展类加载器的 parent 父加载器是 null,并不是引导类加载器。

分析完扩展类加载器,再看系统类加载器的创建,其原理大同小异,需要注意的是,在 Launcher 类构造函数中实例化系统类加载器时,将刚刚创建的扩展类加载器作为入参传递给了 AppClassLoader.getAppClassLoader(final ClassLoader)接口,该接口最终会将系统类加载器的 parent 属性设置为扩展类加载器。因此系统类加载器的父加载器是扩展类加载器,但是扩展类加载器的父加载器是 null。

既然系统类和扩展类加载器都是在 Launcher 类的构造函数中才得以创建,那么 Launcher 类是在什么时间点被实例化呢?这是一个问题,该问题后文会讲。

JVM 在启动过程中,除了会进行扩展类加载器与系统类加载器的实例化,也会进行引导类加载器的初始化。引导类加载器便是上文所展示的使用 C++语言编写的 ClassLoader 类,该类提

供了 `initialize()` 接口, 该接口在 JVM 的 `init()` 初始化链路中被调用, 该接口会读取引导类路径, 定位到相关的 jar 文件, 为加载核心类库做准备。

10.4.2 系统类加载器与扩展类加载器创建

上文讲过, 系统类加载器与扩展类加载器在 `sun.misc.Launcher` 的构造函数中被创建, 但是 `sun.misc.Launcher` 类又在何时被创建呢? 这要从 Java 主类的加载说起。前面讲过, JVM 加载 Java 主类时, 会走下面这条链路:

```
java.c::JavaMain()
  java.c::LoadClass()
    jni.cpp::JNI_ENTRY(jclass, jni_FindClass(JNIEnv *env, const char *name))
    jni.cpp::JNI_ENTRY(jclass, jni_FindClass(JNIEnv *env, const char *name))
    jvm.cpp::find_class_from_class_loader()
      SystemDictionary::resolve_or_fail()
        // .....
        jvm.cpp::jvm_define_class_common()
          SystemDictionary.cpp::resolve_from_stream()
            ClassFileParser.cpp::parseClassFile()
```

这条链路的第 3 步会进入 `jni_FindClass()` 入口, 当 JVM 加载 Java 应用程序主类时, 该入口最终会调用 `loader = Handle(THREAD, SystemDictionary::java_system_loader())` 来获取类加载器, `sun.misc.Launcher` 类的实例便会在 `SystemDictionary::java_system_loader()` 链路里创建。

这里的 `SystemDictionary::java_system_loader()` 返回 `SystemDictionary.java_system_loader`, 这便是传说中的“系统类加载器”。`SystemDictionary.java_system_loader` 成员变量在 JVM 启动期间通过调用 `SystemDictionary::compute_java_system_loader()` 函数进行初始化, 该函数实现如下:

清单: `/src/share/vm/classfile/systemDictionary.cpp`

功能: `compute_java_system_loader()` 函数实现

```
void SystemDictionary::compute_java_system_loader(TRAPS) {
    KlassHandle system_class(THREAD, WK_KLASS(ClassLoader_klass));
    JavaValue result(T_OBJECT);
    JavaCalls::call_static(&result,
                          KlassHandle(THREAD, WK_KLASS(ClassLoader_klass)),
                          vmSymbols::getSystemClassLoader_name(),
                          vmSymbols::void_classloader_signature(),
                          CHECK);
    _java_system_loader = (oop)result.get_jobject();
}
```

该函数通过调用 `JavaCalls::call_static()` 函数来完成 `_java_system_loader` 变量的初始化。

JavaCalls::call_static()函数在 JVM 内部也是调用频率很高的一个接口，大凡涉及 Java 类静态方法调用时，最终都会经过本接口调用。该接口顾名思义，主要作用是调用 Java 类的静态方法，该接口的第 1~4 个人参含义分别如下：

- ◎ JavaValue* result，储存调用 Java 类静态方法后的返回值。
- ◎ KlassHandle klass，所调用的目标 Java 类在 JVM 内部的对等结构体。
- ◎ Symbol* name，所调用的目标 Java 类静态方法的名称。
- ◎ Symbol* signature，目标 Java 类静态方法的签名。

众所周知，Java 程序的方法一定被封装在 Java 类中，所以要调用一个 Java 方法，必定要知道类对象、类名和方法签名，静态方法也不例外，由此便能理解为何调用 JavaCalls::call_static()函数需要传递上面这几个人参。

SystemDictionary::compute_java_system_loader()函数调用 JavaCalls::call_static()函数时，所传入的第 2 个参数是 KlassHandle(THREAD, WK_KLASS(ClassLoader_klass))，其中，ClassLoader_klass 在 systemDictionary.hpp 中通过宏定义，如下：

```
template(ClassLoader_klass, java_lang_ClassLoader, Pre)
```

其实这里所定义的宏比较长，上面这一行仅仅是其中一行，这一点在后面讲解“预加载”的时候再描述。总之，通过这个宏可以知道，ClassLoader_klass 最终映射到的 Java 类是 java.lang.ClassLoader 类。由此可知，JavaCalls::call_static()函数最终要调用的目标方法所在的类是 java.lang.ClassLoader 类。

SystemDictionary::compute_java_system_loader()函数调用 JavaCalls::call_static()函数时，所传入的第 3 个参数是 vmSymbols::getSystemClassLoader_name()，在 vmSymbols.hpp 中，通过下面这个宏定义了 getSystemClassLoader_name()函数的返回值：

```
template(getSystemClassLoader_name, "getSystemClassLoader")
```

由此可知，getSystemClassLoader_name()函数返回“getSystemClassLoader”这个字符串标识。而该标识将作为 JavaCalls::call_static()函数的第 3 个人参，该入参正是 JavaCalls::call_static()函数最终要调用的目标方法。综合上面第 2 和第 3 这两个人参可知，SystemDictionary::compute_java_system_loader()将通过调用 java.lang.ClassLoader 类的静态方法 getSystemClassLoader()来获取类加载器，最终 JVM 将使用这个加载器去加载 Java 程序的主类。

java.lang.ClassLoader.getSystemClassLoader()函数主要通过调用 java.lang.ClassLoader.initSystemClassLoader()接口来初始化系统类加载器，initSystemClassLoader()接口的主要逻辑如下：

清单：/src/java/lang/ClassLoader

功能：系统类加载器初始化

```
private static synchronized void initSystemClassLoader() {
    if (!sclSet) {
        if (scl != null)
            throw new IllegalStateException("recursive invocation");
        sun.misc.Launcher l = sun.misc.Launcher.getLauncher();
        if (l != null) {
            Throwable oops = null;
            scl = l.getClassLoader();
            try {
                scl = AccessController.doPrivileged(
                    new SystemClassLoaderAction(scl));
            } catch (PrivilegedActionException pae) {
                // ...
            }
            if (oops != null) {
                // ...
            }
        }
        sclSet = true;
    }
}
```

在这个函数中，终于看到了 `sun.misc.Launcher` 类的实例化，其实例化通过调用 `sun.misc.Launcher.getLauncher()` 得以完成，该函数直接返回 `Launcher.launcher` 静态变量，而该静态变量在定义时便实例化了，如下：

```
private static Launcher launcher = new Launcher();
```

而系统类加载器和扩展类加载器都是在 `Launcher` 类的构造函数中被创建的，由此 JVM 便完成了系统类加载器和扩展类加载器的创建。

10.4.3 双亲委派机制与破坏

凡是接触过 Java 类加载器的小伙伴，必定知道 JVM 中的双亲委派机制。该机制在 `java.lang.ClassLoader.loadClass(String, boolean)` 接口中，该接口实现在前文讲解 JVM 加载 Java 主类时贴出来过，为了节省篇幅，这里不再贴出了，有不清楚的小伙伴可回头找码看。该接口的逻辑如下：

- (1) 先在当前加载器的缓存中查找有无目标类，如果有，直接返回。
- (2) 判断当前加载器的父加载器是否为空，如果不为空，则调用 `parent.loadClass(name, false)`

接口进行加载。

(3) 反之, 如果当前加载器的父类加载器为空, 则调用 `findBootstrapClassOrNull(name)` 接口, 让引导类加载器进行加载。

(4) 如果通过以上 3 条路径都没能成功加载, 则调用 `findClass(name)` 接口进行加载。该接口最终会调用 `java.lang.ClassLoader` 接口的 `define*` 系列的 `native` 接口加载目标 Java 类。

双亲委派模型就隐藏在这第 2 和第 3 步中。在理解双亲委派机制之前, 有一点需要先说明, 那就是 JVM 中的所有类加载都会通过 `java.lang.ClassLoader.loadClass(String)` 接口 (自定义类加载器并重写 `java.lang.ClassLoader.loadClass(String)` 接口的除外), 连 JDK 的核心类库也不能例外, 虽然核心类库会存在“预加载”的行为, 这一点下文会详细分析。知道了这一点, 方能理解双亲委派模型。假设当前加载的是 `java.lang.Object` 这个类, 很显然, 该类属于 JDK 中核心得不能再核心的一个类, 因此一定只能由引导类加载器进行加载。当 JVM 准备加载 `java.lang.Object` 时, JVM 默认会使用系统类加载器去加载, 按照上面 4 步加载的逻辑, 在第 1 步从系统类的缓存中肯定查找不到该类, 于是进入第 2 步。由于从系统类加载器的父加载器是扩展类加载器, 于是扩展类加载器继续从第 1 步开始重复。由于扩展类加载器的缓存中也一定查找不到该类, 因此进入第 2 步。扩展类的父加载器是 `null` (前文刚讲过), 因此系统调用 `findClass(String)`, 最终通过引导类加载器进行加载。这便是双亲委派机制, 这种机制保证核心类库一定是由引导类加载器进行加载, 而不会被多种加载器加载, 否则每个加载器都会加载一遍核心类库, 世界要大乱了, 同时也会存在安全隐患。

双亲委派从本质上而言, 其实规定了类加载的顺序是: 引导类加载器先加载, 若加载不到, 由扩展类加载器加载, 若还加载不到, 才会由系统类加载器或自定义的类加载器进行加载。

不过, 可能聪明的你会想到, 如果在自定义的类加载器中重写 `java.lang.ClassLoader.loadClass(String)` 或 `java.lang.ClassLoader.loadClass(String, boolean)` 方法, 抹去其中的双亲委派机制, 仅保留上面这 4 步中的第 1 步与第 4 步, 那么是不是就能够加载核心类库了呢? 这也不行! 因为 JDK 还为核心类库提供了一层保护机制。不管是自定义的类加载器, 还是系统类加载器抑或扩展类加载器, 最终都必须调用 `java.lang.ClassLoader.defineClass(String, byte[], int, int, ProtectionDomain)` 方法, 而该方法会执行 `preDefineClass()` 接口, 该接口中提供了对 JDK 核心类库的保护, 其实现如下:

清单: `/src/java/lang/ClassLoader`

功能: 核心类库保护

```
private ProtectionDomain preDefineClass(String name, ProtectionDomain pd) {
    if (!checkName(name))
        throw new NoClassDefFoundError("IllegalName: " + name);
}
```

```

    if ((name != null) && name.startsWith("java.")) {
        throw new SecurityException
            ("Prohibited package name: " + name.substring(0,
name.lastIndexOf('.')));
    }
    if (pd == null) {
        pd = defaultDomain;
    }

    if (name != null) checkCerts(name, pd.getCodeSource());

    return pd;
}

```

在该接口中会判断所要加载的目标类的全限定名是否以“java.”开始，如果是，则直接抛出异常。这便是所有的自定义类加载器都只能重写 `java.lang.ClassLoader.findClass(String)` 接口的原因。

10.4.4 预加载

JVM 启动期间，会先加载一部分核心类库，这部分核心类库包括：

清单：/src/share/vm/classfile/systemDictionary.hpp

功能：定义预加载的类

```

#define WK_KLASSES_DO(template)
/* well-known classes */
template(Object_class,      java_lang_Object,      Pre) \
template(String_class,      java_lang_String,      Pre) \
template(Class_class,       java_lang_Class,       Pre) \
template(Cloneable_class,   java_lang_Cloneable,   Pre) \
template(ClassLoader_class, java_lang_ClassLoader, Pre) \
template(Serializable_class, java_io_Serializable, Pre) \
template(Thread_class,      java_lang_Thread,      Pre) \
template(ThreadGroup_class, java_lang_ThreadGroup, Pre) \
template(Properties_class,   java_util_Properties, Pre) \
template(reflect_AccessibleObject_class, java_lang_reflect_AccessibleObject, Pre) \
template(reflect_Field_class, java_lang_reflect_Field, Pre) \
template(reflect_Method_class, java_lang_reflect_Method, Pre) \
template(Float_class,        java_lang_Float,        Pre) \
template(Double_class,        java_lang_Double,        Pre) \
template(Byte_class,         java_lang_Byte,         Pre) \
template(Short_class,        java_lang_Short,        Pre) \
template(Integer_class,      java_lang_Integer,      Pre) \
template(Long_class,         java_lang_Long,         Pre) \

```



```
// ...
/*end*/
```

可以看到,平时开发中最常用的基础类库,基本都在这里了,例如 `Object`、`Long`、`String`、`Serializable` 及 `Thread` 等。也正是因为这些类使用频率非常高,即使是一个非常简单的 Java 程序都可能用到这些类中的大部分类,因此不如预先将其加载到内存。但是同时也应该看到,放眼 JDK 的整个核心类库,这几个类也是凤毛麟角,这是因为预加载的类越多,则 JVM 的启动过程将越慢,反而不美。

核心类库的加载是可以被观察到的,只需要在 Java 命令行上加上 `-XX:+TraceClassLoading` 选项, JVM 启动时便会打印类似下面这样的跟踪类加载的日志:

```
[Opened
/Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded java.lang.Object from
/Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded java.io.Serializable from
/Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded java.lang.Comparable from
/Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded java.lang.CharSequence from
/Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/jre/lib/rt.jar]
[Loaded java.lang.String from
/Library/Java/JavaVirtualMachines/jdk1.8.0_60.jdk/Contents/Home/jre/lib/rt.jar]
// ...
```

JDK 核心类库随 JVM 启动而进行预加载的链路如下:

```
java.c::main()
  java_md.c::LoadJavaVM()
    jni.cpp::JNI_CreateJavaVM()
      Threads::create_vm()
        init.cpp::init_globals()
          Universe.cpp::universe2_init()
            Universe::genesis()
              SystemDictionary::initialize()
                SystemDictionary::initialize_preloaded_classes()
                  SystemDictionary::initialize_wk_klasses_until()
```

在 `SystemDictionary::initialize_wk_klasses_until()` 函数中,遍历 `WK_KLASSES_DO` 宏中所定义的全部预加载的类,并调用 `SystemDictionary::initialize_wk_class()` 函数对这些类进行逐个加载。`SystemDictionary::initialize_wk_class()` 函数实现如下:

清单: `/src/share/vm/classfile/systemDictionary.cpp`

功能: JDK 预加载核心类库

```
bool SystemDictionary::initialize_wk_class(WKID id, int init_opt, TRAPS) {
```



```

// ...
if ((*klassp) == NULL && try_load) {
    if (must_load) {
        (*klassp) = resolve_or_fail(symbol, true, CHECK_0); // load required class
    } else { // ... }
}
return ((*klassp) != NULL);
}

```

在该函数中，最终调用 `SystemDictionary::resolve_or_fail()` 接口去加载核心类库。在前文分析 JVM 加载 JAVA 主类时，其链路也会经过这里，该函数会一路调用，最终调用到 `SystemDictionary::load_instance_class()` 函数。到了这个函数中，开始体现出“双亲委派”的机制。该函数根据类加载器变量 `class_loader` 是否为 `null`，被分成了 `if{} 和 else{} 这两大分支`，如果是 `null` 则进入 `if{} 逻辑块`，否则进入 `else{} 逻辑块`。由于 JVM 在启动期间加载核心类库时，核心类库压根儿没有所谓的“类加载器”概念，因此核心类库的预加载流程直接进入 `if{} 逻辑块`，而前文在分析 JVM 加载 Java 应用程序主类时，其链路进入了 `else{} 逻辑块`。在 `if{} 逻辑块` 中，JVM 会调用内部的引导类加载器来加载核心类库，而在 `else{} 逻辑块` 中，则会通过 `JavaCalls::call_virtual()` 接口来加载类，而这个接口最终会通过 `java.lang.ClassLoader.loadClass(String)` 接口执行类的加载。这便是 jvm 内部的“双亲委派”机制，一边是直接使用引导类加载器，一边则是使用系统类加载器。在 `if{} 语句块` 中，可以看到 JVM 调用了 `ClassLoader::load_classfile()` 接口，这是引导类加载器的加载接口。

由于 JDK 核心类库的预加载直接使用了引导类加载器，不经过 `java.lang.ClassLoader.loadClass(String)` 接口，因此如果在这个接口内部打断点，是无法看到核心类库的加载过程的。

正是由于 JVM 启动期间会对部分核心类库进行预加载，因此 JVM 在加载 Java 应用程序主类的链路中对 `sun.misc.Launcher` 类进行初始化时，才能直接获取到 `java.lang.ClassLoader` 类在 JVM 内部对应的实例对象，并调用其静态方法 `getSystemClassLoader()` 来初始化系统类加载器和扩展类加载器。

10.4.5 引导类加载

JVM 使用“双亲委派”机制加载类，如果所加载的类属于 JDK 核心类库中所定义的类，则 `java.lang.ClassLoader.loadClass(String, boolean)` 会进入 `findBootstrapClassOrNull()` 方法，通过引导类加载核心类库。该方法最终会调用 `private native Class<?> findBootstrapClass(String name)` 这个本地接口执行核心类库加载。这个本地接口最终会调用引导类加载器执行加载。开放的 JDK 源

码中包含这个本地接口的实现，如下（仅摘录核心逻辑）：

清单：/src/share/native/java/lang/ClassLoader.c

功能：JNI 本地接口加载核心类库接口

```

JNIEXPORT jclass JNICALL
Java_java_lang_ClassLoader_findBootstrapClass(JNIEnv *env, jobject loader,
                                              jstring classname)
{
    char *cname;
    jclass cls = 0;
    char buf[128];

    cname = getUTF(env, classname, buf, sizeof(buf));
    cls = JVM_FindClassFromBootLoader(env, cname);
    return cls;
}

```

在 JNI 实现中，调用 JVM_FindClassFromBootLoader() 来加载核心类库。而在 JVM_FindClassFromBootLoader() 接口中，则直接调用 SystemDictionary::resolve_or_null() 函数，该函数最终也会走到 SystemDictionary::load_instance_class() 函数中。关于该函数，前面多次提到过，无论是 JVM 在启动期间预加载部分核心类库还是 JVM 加载 Java 应用程序主类，只要涉及类的加载，最终都会经过该函数。而在上文也提到过，该函数内部也使用了“双亲委派”机制，如果 class loader 为空，则直接调用引导类加载器的 ClassLoader::load_classfile() 接口进行类加载，否则，仍然要通过所指定的 Java 类加载器去加载。由于在加载核心类库时，类加载器为空，因此核心类库的加载最终都会通过调用引导类加载器接口进行加载。

10.4.6 加载、链接与延迟加载

看下面这个测试程序：

清单：/Test.java

功能：核心类库加载调试

```

public class Test {
    public static void main(String[] args) {
        java.lang.Long ll = 3000L;
    }
}

```

在 java.lang.ClassLoader.findBootstrapClassOrNull(String) 中打上断点，调试时会发现 java.lang.Long 这个核心类仍然会经过该断点，这说明 java.lang.Long 核心类仍然通过 java.lang.ClassLoader 进行加载。但是前面讲过，JVM 在启动时会预加载一部分核心类库，包括

java 基本类型的包装类，因此 java.lang.Long 这种核心类也一定早就预加载好了，可是为何在测试程序中实例化 Long 类型时，仍然要再次加载一遍呢？其实，之所以会再次走加载流程，并不是因为碰到了类加载的指令，而是因为测试类所对应的常量池索引尚未转换成直接对象引用，更准确地说，是因为预加载的核心类库虽然已经完成了类的加载，但是由于尚未在应用程序中使用到，因此并未经过链接，所以在 new 字节码指令的流程中便走了“慢分配”流程，这在后文讲解类实例化机制时会讲到。

众所周知，Java 类的生命周期一共分为 7 个阶段，其中加载阶段仅仅是其中第一步，加载完成之后需要进行链接和初始化。在链接阶段，字节码指令会被重写，将其所引用的常量池的索引号转换为直接引用。例如，在实例化一个类时，编译后所生成的字节码指令如下：

```
new #2
```

new 指令后面跟随的#2 表示引用常量池中索引号为 2 的元素，该元素一定指向某个 Java 类的全限定名。如果是实例化 Long，则常量池 2 号索引指向的字符串一定如下：

```
class java/lang/Long
```

在 Java 类经编译后所得到的字节码文件中的原始常量池中，class java/lang/Long 并不是存放于常量池的一个元素中而是分开存放于好几个元素中，例如，class 本身是常量池中的一种固定的类型，而 java/lang/Long 在常量池中则是一种字符串类型，UTF-8 格式的字符串。当 JVM 加载完测试类 Test 之后，会对其字节码进行重写，重写后的 new 字节码指令，其后面所跟随的便是直接指向“java/lang/Long”这个字符串的内存地址，这个重写便是整个 Java 类生命周期中“链接”阶段所做的最重要的事情。其实重写字节码的过程，可以认为是做了一次缓存的优化，通过重写，避免运行期再去将多个不同的常量池元素一步步拼装出 new 指令实际要指向的类型。

等到 JVM 真正运行到 new 这条字节码指令时，JVM 为了加快指令执行速度，又做了一次缓存。这一次缓存的是啥呢？众所周知，在 Java 语言中，new 指令的作用很简单也很唯一，就是实例化一个类型，或者数组。当使用 new 实例化一个 Java 类型时，JVM 必须要知道其所实例化的是何种类型，这个问题在 Java 类链接阶段便已解决了，通过字节码指令重写，JVM 知道所要实例化的 Java 类的全限定名。但是仅仅这样仍然不够，放眼 JVM 执行 new 指令的过程，JVM 需要根据 Java 类的全限定名称，在内存 perm 区（JDK 8 中是 metaspace 区）定位到这个 Java 类在内存中的对等体——instanceKlass，instanceKlass 作为 Java 类在内存中的对等体，包含了原始 Java 类中的一切信息，JVM 只有找到了 instanceKlass，才能根据这个类模板创建出 Java 类的实例对象。在 JVM 开始执行 new 指令所对应的实例对象创建过程之前，一定会先完成类的加载、链接和初始化，instanceKlass 便在类的加载阶段完成构建，因此 JVM 根据 Java 类的全限定名称一定能够定位到 instanceKlass 这个类模板。但是如果每次执行 new 指令都要根据 Java 类的全限定名称去定位到 instanceKlass 这个内存对象，未免会做重复的事情，浪费机器性能，

为了避免这种情况, JVM 在第一次执行 new 指令时, 便会将定位到的 instanceClass 缓存起来, 这样如果程序后续需要再次实例化同样的 Java 类对象时, 便直接从缓存中读取 instanceClass, 直接据此创建 Java 类实例对象, 从而提升效率。前文曾提及, 使用 new 指令加载类时, 如果类尚未被加载或者未被链接过, 则会进入慢分配流程(后文会细讲类分配机制), 从而会进入 InterpreterRuntime::_new()函数, 而 InterpreterRuntime::_new()函数则会调用 constantPool->class_at()接口来获取 new 指令后面的 Java 类模板, 该接口最终调用 constantPoolOopDesc::class_at_impl()函数, 该函数实现如下(仅摘录主要逻辑):

清单: /src/share/vm/oops/constantPoolOop.cpp

功能: Java 类模板缓存

```

classOop constantPoolOopDesc::class_at_impl(constantPoolHandle this_oop, int
which, TRAPS) {
    // 从常量池中取指定位置的元素
    CPSlot entry = this_oop->slot_at(which);

    // 第一次执行 new 指令时, 不会进入该分支
    if (entry.is_oop()) {
        return (klassOop)entry.get_oop();
    }

    // ...
    { ObjectLocker ol(this_oop, THREAD);
      // 获取类加载器
      loader = Handle(THREAD, instanceClass::cast(this_oop->pool_holder())->
class_loader());
    } // unlocking constantPool

    // ...
    if (do_resolve) {
        // 执行类加载
        klassOop k_oop = SystemDictionary::resolve_or_fail(name, loader, h_prot,
true, THREAD);

        // ...
        if (TraceClassResolution && !k()->class_part()->oop_is_array()) {
            // ...本逻辑块处理 new 数组的情况
        } else { // 如果不是实例化数组, 则一定是 Java 类

            // 再次判断目标类是否已缓存, 可能其他 Java 线程会先于本线程执行 new 操作
            do_resolve = this_oop->tag_at(which).is_unresolved_class();

            if (do_resolve) {
                // 将目标类模板缓存到常量池中, 下次执行 new 指令时直接从这里取
            }
        }
    }
}

```

```

        this_oop->klass_at_put(which, k());
    }
}
// 从常量池指定位置的元素中获取类模板
return (klassOop)entry.get_oop();
}

```

上面这段代码的主要思路是，先从缓存中读取 Java 类模板，如果读取不到，则执行类加载，加载之后，将类模板写入缓存，最终仍然从缓存中读取类模板并返回。这与 java 应用程序中的缓存读写逻辑何其相似！

在该逻辑中，如果目标 Java 类模板尚未被解析，则 JVM 会调用 `SystemDictionary::resolve_or_fail()` 接口先执行类加载。前文已经讲过该接口所经过的链路，该接口最终会调用 `SystemDictionary::load_instance_class()` 函数执行类加载，对于 `SystemDictionary::load_instance_class()` 函数，前面分析过其实现机制，其内部也使用了一个类似于“双亲委托”的机制。当类加载器为 null 时，则直接加载核心类库，否则最终仍然会调用 `java.lang.ClassLoader.loadClass(String)` 这个 Java 接口去执行类加载。由于在本示例程序中，通过执行 `new` 指令进入了 `SystemDictionary::load_instance_class()` 函数，而 Java 程序的默认类加载器便是系统类加载器，因此 `SystemDictionary::load_instance_class()` 函数最终仍然通过 `java.lang.ClassLoader.loadClass(String)` 这个 Java 接口去执行类加载。这是为何在 `java.lang.ClassLoader.findBootstrapClassOrNull(String)` 中打上断点，而调试时会发现 `java.lang.Long` 这个核心类仍然会经过该断点进行加载的原因。

不过前面也讲过，`java.lang.Long` 在 JVM 启动期间便被预加载，因此最终通过 `java.lang.ClassLoader.findBootstrapClassOrNull()` 接口调用引导类加载器去加载核心类库时，JVM 内部并未真的重新将 `java.lang.Long` 加载一遍，而是直接从缓存中读取。

将上面的测试示例稍微修改一下，变成如下：

清单：/Test.java

功能：核心类库加载调试

```

public class Test {
    public static void main(String[] args) {
        java.lang.Long l1 = 3000L;
        java.lang.Long l11 = 5113L;
    }
}

```

在本示例程序中，连续两次实例化了 `Long` 类型，在 `java.lang.ClassLoader`

`findBootstrapClassOrNull(String)`中打上断点，调试时会发现 `java.lang.Long` 这个核心类的加载仅在执行本示例程序的第一个 `Long` 实例化代码时才会经过该断点，而第二次实例化 `Long` 类型时则不会再经过该断点，道理其实很简单，因为第一次实例化之后，`java.lang.Long` 这个类模板已经完成解析，所以第二次再次实例化时，JVM 便会尝试走“快速分配”流程进行无锁分配，如果该类不满足快速分配的条件（例如类太大），虽然仍然会进入慢分配流程，但是由于已经解析过，因此在慢分配阶段并不会经历类加载过程。

为了证明在对 JDK 核心类库中被预加载的类执行 `new` 实例化操作时，JVM 没有重复加载类，在断点测试时开启 `-XX:+TraceClassLoading` 选项，查看本示例程序中的 `Long` 类型是何时被加载的。

事实上，不仅 JDK 核心类的加载如此，所有的 Java 类的实例化都是如此，各位道友可以自由编写测试程序进行断点调试。很多书籍和网络博客都说 Java 类的加载、链接和初始化并没有严格的顺序，可以混着来，其实道理便在这里。

从更广义的角度看，其实这也体现了类的延迟加载机制——一个类，只有当真正被使用时才会被加载，即使程序中 `import` 了某个类，但是如果不使用，则 JVM 在整个运行期都不会加载它。甚至，即使在程序中使用到了某个类，但是如果使用条件一直不符合，导致 JVM 一直没有进入使用的分支流程中，则 JVM 也自始至终都不会加载这个类。例如下面这段示例代码：

```
if(...){
    new Test();
}
```

如果整个程序只有这一处地方用到了 `Test` 类，但是程序在运行期从来都没有进入过这里的 `if` 分支流，则 `Test` 类不会被加载。不用说，延迟加载机制避免了系统无谓的开销。

10.4.7 父加载器

前面讲过，JVM 默认提供 3 种类加载器：引导类加载器、扩展类加载器和系统类加载器。其中，系统类加载器的父加载器是扩展类加载器，而扩展类加载器与引导类加载器的父加载器都是 `null`，这是 JVM 中固化下来的关系设定。默认情况下，Java 应用程序的类加载器是系统类加载器。

下面这个测试程序可以验证父加载器的相关理论：

清单：/Test.java

功能：父加载器

```
public class Test {
    public static void main(String[] args) throws Exception {
```

```
Test t = new Test();

System.out.println("Test.classLoader is " + t.getClass().getClassLoader());
System.out.println("Test.classLoader.parentClassLoader is "
    + t.getClass().getClassLoader().getParent());

System.out.println("Test.classLoader.parentClassLoader.parentClassLoader is "
    + t.getClass().getClassLoader().getParent().getParent());
}
```

运行后, 打印结果如下:

```
Test.classLoader is sun.misc.Launcher$AppClassLoader@2d3fcd9d
Test.classLoader.parentClassLoader is sun.misc.Launcher$ExtClassLoader@7225790e
Test.classLoader.parentClassLoader.parentClassLoader is null
```

根据该结果可知, Test 测试类的加载器是 Launcher\$AppClassLoader, 而这正是系统类加载器。同理, Test 测试类加载器的父加载器是 Launcher\$ExtClassLoader, 这是扩展类加载器。通过这里的打印结果可知, 扩展类的父加载器确实是 null。而至于 Test 类的加载器为何是系统类加载器, 前文从源码级别进行了分析。

前面讲过, JDK 核心类库中的类由引导类加载器负责加载, 那么如果在测试程序中加载一个核心类, 然后打印其加载器, 会打印出啥呢? 测试程序很简单, 如下:

清单: /Test.java

功能: 父加载器

```
public class Test {
    public static void main(String[] args) throws Exception {
        Object obj = new Object();
        System.out.println("Object.classLoader is " +
            obj.getClass().getClassLoader());
    }
}
```

该测试程序十分简单, 仅仅实例化了一个 java.lang.Object 类。该类绝对是 JDK 核心类库中的核心类。

打印结果是:

```
Object.classLoader is null
```

不是说好核心类库的加载器是引导类加载器吗, 为何这里却打印出一个空值呢? 道理其实很简单, 站在程序的角度看, 引导类加载器与另外两种类加载器——系统类加载器和扩展类加载器, 并不是同一个层次意义上的加载器, 引导类加载器是使用 C++ 语言编写而成的, 而另外

两种类加载器则是使用 Java 语言编写而成的。由于引导类加载器压根儿就不是一个 Java 类，因此在此 Java 程序中只能打印出空值。

JVM 提供自定义类加载器的接口，开发者只需继承 `java.lang.ClassLoader` 或者其子类（例如 `java.net.URLClassLoader`），重写相关接口，便能自定义类加载器。例如下面的示例：

清单：/Test.java

功能：自定义类加载器

```
public static void main(String[] args) throws Exception {
    // 自定义类加载器
    ClassLoader loader = new ClassLoader() {
        @Override
        public Class<?> loadClass(String name) throws ClassNotFoundException {
            try {
                String className =
                    name.substring(name.lastIndexOf(".") + 1) + ".class";
                InputStream is = getClass().getResourceAsStream(className);
                if (is == null) {
                    return super.loadClass(name);
                }
                byte[] buffer = new byte[is.available()];
                is.read(buffer);
                return defineClass(name, buffer, 0, buffer.length);
            } catch (Exception e) {
                throw new ClassNotFoundException(name);
            }
        }
    };

    // 使用自定义类加载器加载 Test 测试类
    Object test = loader.loadClass("Test").newInstance();
    System.out.println("Test.classLoader is " + test.getClass().
        getClassLoader());
    System.out.println("Test.classLoader.parentClassLoader is " +
        test.getClass().getClassLoader().getParent());

    System.out.println("Test.classLoader.parentClassLoader.parentClassLoader is " +
        test.getClass().getClassLoader().getParent().getParent());
}
```

运行该测试程序，打印结果如下：

```
Test.classLoader is Test$1@36f6e879
Test.classLoader.parentClassLoader is sun.misc.Launcher$AppClassLoader@2d3fcd8d
Test.classLoader.parentClassLoader.parentClassLoader is
sun.misc.Launcher$ExtClassLoader@531be3c5
```

通过打印结果可知，Test 类的加载器是 Test\$1@36f6e879，这里的 Test\$1 表示是 main 主函数中的局部变量，因为在本示例程序中，自定义的加载器被定义成局部变量。接着看这个自定义类加载器的父加载器，是 Launcher\$AppClassLoader，即系统类加载器。这里比较奇怪，明明自定义的类加载器直接继承了 java.lang.ClassLoader，并没有继承 Launcher\$AppClassLoader，但是为何自定义的类加载器的父加载器变成了系统类加载器呢？原因其实很简单，这个秘密隐藏在 java.lang.ClassLoader 的默认构造函数中，其默认的无参构造函数是：

清单：/src/java/lang/ClassLoader.java

功能：java.lang.ClassLoader 的默认构造函数

```
protected ClassLoader() {
    this(checkCreateClassLoader(), getSystemClassLoader());
}
```

通过这个默认构造函数可知，如果自定义的类加载器没有重写构造函数，则该默认构造函数会将 getSystemClassLoader() 返回的结果作为自定义类加载器的父加载器。而 getSystemClassLoader() 函数在前文已经讲过，最终返回的便是系统类加载器——Launcher\$AppClassLoader。

10.4.8 加载器与类型转换

经过上述分析可知，每个 Java 类都必定有一个类加载器，JDK 核心类库的加载器是引导类加载器，应用程序中的类加载器默认是系统类加载器，除此以外，开发者还可以为应用程序开发自定义的加载器。例如 Tomcat 这类 Web 应用服务器，内部自定义了好几种类加载器，用于隔离同一个 Web 应用服务上的不同应用程序。

在一般情况下，使用不同的类加载器去加载不同的功能模块，会提高应用程序的安全性。但是，如果涉及 Java 类型转换，则加载器反而容易产生不美好的事情。在做 Java 类型转换时，只有两个类型都是由同一个加载器所加载，才能进行类型转换，否则转换时会发生异常。将上面自定义类加载器的那个示例稍微修改一下，变成如下：

清单：/Test.java

功能：自定义类加载器与类型转换

```
public static void main(String[] args) throws Exception {
    // 自定义类加载器
    ClassLoader loader = new ClassLoader() {
        @Override
        public Class<?> loadClass(String name) throws ClassNotFoundException {
            try {
                // ...
            } catch (Exception e) {
```

```

        throw new ClassNotFoundException(name);
    }
}

// 使用自定义类加载器加载 Test 测试类
Test test = (Test)loader.loadClass("Test").newInstance();
System.out.println("Test.classLoader is " + test.getClass().
getClassLoader());
System.out.println("Test.classLoader.parentClassLoader is " +
test.getClass().getClassLoader().getParent());

System.out.println("Test.classLoader.parentClassLoader.parentClassLoader is " +
test.getClass().getClassLoader().getParent().getParent());
}

```

这里修改了一行代码，原本在加载 Test 实例时，没有做类型转换，直接返回 Object 类型，而这里进行了类型转换。运行程序，最终会抛出下面这条异常：

```
Exception in thread "main" java.lang.ClassCastException: Test cannot be cast to Test
```

异常信息提示得很清楚：Test 类无法转换成 Test 类。乍一看，这个异常抛得莫名其妙，同一个类型竟然无法转换。但是实际上 JVM 并没有错，错的是你。在 `Test test = (Test)loader.loadClass("Test").newInstance()` 这句代码中，等号左边所声明的 Test 类型，由于测试程序并没有明确为其指定类加载器，因此 JVM 会使用系统类加载器加载 Test 类。而等号右边则明确使用了自定义的类加载器加载 Test 类型。因此等号左边与右边的两个 Test 类型的加载器并不是同一个，所以程序便抛出异常了。

ClassCastException 这类异常在使用 Spring 框架的 Java 应用程序中比较常见，相信很多道友都遇到过。究其原因，还是因为很多中间件内部都有自定义的类加载器，因此被内存加载器所加载的类型，无法直接转换为使用默认加载器加载的类型。

10.5 类实例分配

编写 Java 程序，使用最频繁的指令几乎就是 new 了，因为 Java 所有的数据和行为都封装在类中，想要读写数据或者触发某种行为，必须先实例化 Java 类。实例化 Java 类的方式有很多，例如，调用 `java.lang.Class.newInstance()` 或者直接使用 new 指令。前文在讲解类加载过程时对 new 的内部机制进行了局部介绍，而事实上，new 的实现机制相当精彩，对内存、线程并发控制等几乎都达到了登峰造极的地步，利用了软件和硬件所能利用的一切优化手段。并且，对象

实例化涉及内存分配，而内存分配又与垃圾收集器紧密耦合在一块，所以可以这么说，这部分的技术实现实在是精华中的精华。相比于执行引擎中对硬件指令的封装、运行期多层动态编译和指令实时优化的精微和深奥，内存分配则显得相当简单——什么技术够高大上，就直接拿来使用，为了性能和内存开销用尽一切谋略。

HotSpot 提供了 new 字节码指令的机器码实现，在 Intel 32 位平台上，其实是在 `templateTable_x86_32.cpp::new()` 函数中。该函数中的代码都是为了在运行期生成硬件相关的机器码。好在 HotSpot 保留了字节码解释器的实现，并且字节码解释器中的实现逻辑与机器码逻辑基本一致，所以为了方便，可以直接参考字节码解释器的逻辑来分析实例化的机制。字节码解释器的实现如下：

清单：/src/share/vm/interpreter/bytecodeInterpreter.cpp

功能：new 指令的实现机制

```
void BytecodeInterpreter::run(interpreterState istate) {
    //...
    run:
        //...

    CASE(_new): {
        u2 index = Bytes::get_Java_u2(pc+1);
        constantPoolOop constants = istate->method()->constants();

        // 如果目标 Java 类已经解析
        if (!constants->tag_at(index).is_unresolved_class()) {
            oop entry = constants->slot_at(index).get_oop();
            klassOop k_entry = (klassOop) entry;
            instanceKlass* ik = (instanceKlass*) k_entry->klass_part();

            // 如果符合快速分配场景
            if ( ik->is_initialized() && ik->can_be_fastpath_allocated() ) {
                size_t obj_size = ik->size_helper();
                oop result = NULL;
                bool need_zero = !ZeroTLAB;
                if (UseTLAB) {
                    // 先通过 tlab 进行分配
                    result = (oop) THREAD->tlab().allocate(obj_size);
                }

                // 如果 tlab 分配失败，则在 eden 区分配
                if (result == NULL) {
                    need_zero = true;
                    // Try allocate in shared eden
                }
            }
            retry:
                HeapWord* compare_to = *Universe::heap()->top_addr();
```

```

// 指针碰撞分配法(bump -the-pointer)
HeapWord* new_top = compare_to + obj_size;
if (new_top <= *Universe::heap()->end_addr()) {
    if (Atomic::cmpxchg_ptr(new_top, Universe::heap()->top_addr(),
compare_to) != compare_to) {
        goto retry;
    }
    result = (oop) compare_to;
}
}
if (result != NULL) {

// tlab 区清零
if (need_zero) {
    HeapWord* to_zero = (HeapWord*) result + sizeof(oopDesc) / oopSize;
    obj_size -= sizeof(oopDesc) / oopSize;
    if (obj_size > 0) {
        memset(to_zero, 0, obj_size * HeapWordSize);
    }
}
if (UseBiasedLocking) {
    result->set_mark(ik->prototype_header());
} else {
    result->set_mark(markOopDesc::prototype());
}
result->set_class_gap(0);
result->set_class(k_entry);

// 将对象地址压入操作数栈栈顶
SET_STACK_OBJECT(result, 0);

// 更新程序计数器 pc, 取下一条字节码指令, 继续处理
UPDATE_PC_AND_TOS_AND_CONTINUE(3, 1);
}
}
}
// 慢分配
CALL_VM(InterpreterRuntime::_new(THREAD, METHOD->constants(), index),
handle_exception);
SET_STACK_OBJECT(THREAD->vm_result(), 0);
THREAD->set_vm_result(NULL);
UPDATE_PC_AND_TOS_AND_CONTINUE(3, 1);
}
//...
}

```

这段逻辑从宏观上分为两部分：一部分是快速分配，一部分则是慢分配。如果所要 new 的 Java 类型尚未被解析过（即使已经被加载也不算），则直接进入慢分配，这便是前文所讲述的 JVM 延迟加载的基础所在。快速分配的流程比较复杂，而慢分配则直接调用 `InterpreterRuntime::_new()` 接口。前文在讲解 Java 类加载时多次提到这个接口。

为了尽可能地加快内存分配速度，并减少并发操作带来的性能损失，JVM 在分配内存时，总是优先使用快速分配策略，当快速分配失败时，才会启用慢分配策略，这便是上面这段源码的总体逻辑，这段逻辑可以概括为如下几点：

- （1）若 Java 类尚未被解析，则直接进入慢分配，不会使用快速分配策略。
- （2）快速分配。如果没有开启栈上分配或不符合条件则会进行 TLAB 分配。
- （3）快速分配。如果 TLAB 分配不成功，则尝试在 eden 区分配。
- （4）如果 eden 区分配失败，则会进入慢分配流程。
- （5）如果对象满足了直接进入老年代的条件，那就直接分配在老年代。

（6）快速分配。对于热点代码，如果开启逃逸分析，JVM 则会执行栈上分配或标量替换等优化方案。

第 6 点中的热点代码的逃逸分析并不包含在本逻辑中，下文会详解。

10.5.1 栈上分配与逃逸分析

前文在讲解 JVM 的 JIT 即时编译器时曾提到过逃逸分析。即时编译（just-in-time compilation, JIT）是一种通过在运行时将字节码翻译为机器码，从而改善字节码编译语言性能的技术。在 HotSpot 中有多种实现：C1、C2 和 C1+C2，分别对应 Client、Server 和分层编译。未来究竟还会有何种更加精妙的实现谁也不说准。

而所谓逃逸，是指一个在方法内部被创建的对象不仅在方法内部被引用，还在方法外部被其他变量引用，这带来的后果是：在该方法执行完毕之后，该方法中创建的对象无法被 GC 回收，因为对象在方法外部还被引用着，这便是逃逸的含义。JVM 所进行的逃逸分析是确定方法内部所创建的对象会不会逃逸出方法体外部，如果确定不会逃逸出去，那么就能对该对象采用多种优化措施，这些优化措施主要围绕两大方面进行：内存分配和线程同步。

逃逸分析的算法主要基于连通图，通过引入连通图来构建对象和对象引用之间的可达性关系，并在此基础上，提出一种组合数据流分析法。该算法是上下文相关和流敏感的，同时模拟了对象任意层次的嵌套关系，所以运行时间比较长和内存消耗比较大（要感知上下文和程序流），但是分析精度比较高。然而其并不能确保百分百的准确性，因为 Java 语言拥有许多动态特性，

例如动态生成字节码、调用本地函数、反射、方法拦截等，这些语言特性导致逃逸分析的算法不能作为编译期间的静态优化措施，而只能是基于运行时的动态分析，而这正是逃逸分析为何需要感知运行时的上下文和程序流的原因。一个对象在编译期可能并没有发生逃逸，但是可能被一个 AOP 框架拦截，结果就不好说了，很可能在运行期就会发生方法逃逸甚至线程逃逸。

由于逃逸分析是在运行期进行的，并且很耗费内存和 CPU 资源，因此 JVM 不可能对每一个方法里的变量都进行逃逸分析，所以其只能作为 JIT 的一项优化措施，即只有 JVM 触发 JIT 编译时才会进行逃逸分析。这也是为何在上面所贴出来的 BytecodeInterpreter::run() 函数源码中并没有看到丝毫与所谓栈上分配相关的实现的原因。在逃逸分析完成之后，JIT 编译器会基于逃逸分析的结果，直接基于 Java 字节码指令，生成优化后的本地机器码指令，所生成的本地机器指令会直接将 Java 对象分配在栈甚至硬件寄存器中。这些优化的本地机器指令已经再也看不到 Java 的 new 字节码指令了，而这也是并不能在 Java 的 new 字节码指令的直接实现里看到栈上分配相关的实现的原因。

清单：Test.java

功能：方法逃逸

```
public class Test {
    int k;

    public static void main(String[] args){
        Test t1 = new Test();
        Test t2 = new Test();
        t2.foo(t1);
    }

    public void foo(Test test){
        Test t = test;
    }

    public void doSomething(){
    }
}
```

在该示例的 main() 主函数中，创建了 Test 类的两个实例 t1 和 t2，其中 t2 并没有在主函数之外被引用，因此不会发生逃逸。而 t1 被传递到了 foo() 方法，不过由于 t2 没有逃逸，因此 JVM 认为 t1 也没有逃逸。

JIT 基于逃逸分析的结果，可以使用不同的策略，为对象实例分配内存，在不同的策略下所生成的本地机器码自然不同。目前主要的优化技术包括标量替换和栈上分配。这两种优化技术都不会将对象实例直接分配在堆上。

1. 标量替换

所谓标量，是指不可分割的量，Java 中的基本数据类型和 reference 类型都属于标量，其中 reference 类型在 JVM 内部其实就是一个指针，因此也属于不可分割的量。如果一个数据类型可以继续分解，则称为聚合量。如果将一个对象拆散，将其成员变量恢复到基本类型以用于访问，这个过程就叫作标量替换。由此可知，标量替换不仅仅是替换那么单纯，替换后还要修改类型字段的读写指令。如果标量替换的优化比较激进，甚至可以直接将一个类的所有字段都打散分配到硬件寄存器中，当然前提是这个类型中包含的字段不能太多，毕竟寄存器数量是有限的。

2. 栈上分配

如果一个类实例引用变量没有发生逃逸，则将实例对象直接分配在方法栈上。在栈上分配的对象实例，会随着 Java 方法执行结束后方法栈空间的被回收而被回收，因此不需要 GC 来回收。这种方式所带来的好处是不言而喻的，毕竟 GC 时间越少，则 JVM 留给用户线程执行的时间片段就越多。

不过栈上分配也有其硬伤，那就是 Java 类型不能太大，包含的字段不能太多，毕竟堆栈空间是有限的，容纳不下几个大型的 Java 类。如果不是因为这个限制，JVM 也不必建立所谓的堆内存空间。

逃逸分析的算法比较复杂，并且与本地机器指令有关，这里就不贴了，毕竟绝大多数开发者都不需要关心其具体实现。但是逃逸分析和基于此所进行的内存分配优化可以通过 jmap -histo 命令观察开启和关闭逃逸分析选项这两种情况下的实例总数来进行验证。除了这种办法，也可以通过观察 GC 日志来间接分析。例如下面这个示例：

清单：Test.java

功能：验证逃逸分析及基于此的内存分配优化

```
public class Test {
    int k = 3;
    long l = 300L;
    int m = 10;

    public static void main(String[] args){
        for(int i = 0; i < 500000; i++){
            Test t = new Test();
        }

        // 可以在这里打断点，通过 jmap -histo 统计开启和关闭
        // 逃逸分析两种情况下的 Test 实例总数
        System.out.println("==1");
    }
}
```

本示例程序中通过一个 50 万次循环，不断创建 Test 类实例对象。之所以要循环，是因为只有达到一定的循环次数才能成为热点代码，才会触发 JIT 编译优化，也才会启用逃逸分析。使用 `-server -Xmx5m -Xms5m -XX:-DoEscapeAnalysis -XX:+PrintGC` 选项运行该测试程序，结果输出如下：

```
[GC (Allocation Failure) 1024K->715K(5632K), 0.0012355 secs]
[GC (Allocation Failure) 1739K->1015K(5632K), 0.0009311 secs]
[GC (Allocation Failure) 2039K->1253K(5632K), 0.0007820 secs]
[GC (Allocation Failure) 2277K->1293K(5632K), 0.0006812 secs]
[GC (Allocation Failure) 2317K->1309K(5632K), 0.0006202 secs]
[GC (Allocation Failure) 2333K->1325K(5632K), 0.0006383 secs]
[GC (Allocation Failure) 2349K->1413K(5632K), 0.0009138 secs]
[GC (Allocation Failure) 2437K->1477K(5632K), 0.0004426 secs]
[GC (Allocation Failure) 2501K->1445K(5632K), 0.0003631 secs]
[GC (Allocation Failure) 2469K->1477K(5632K), 0.0003411 secs]
[GC (Allocation Failure) 2501K->1397K(5632K), 0.0003140 secs]
// ...
```

`-XX:-DoEscapeAnalysis` 选项表示关闭逃逸分析。接着使用 `-server -Xmx5m -Xms5m -XX:+DoEscapeAnalysis -XX:+PrintGC` 选项运行该测试程序，结果输出如下：

```
[GC (Allocation Failure) 1024K->716K(5632K), 0.0011911 secs]
[GC (Allocation Failure) 1740K->962K(5632K), 0.0039492 secs]
[GC (Allocation Failure) 1986K->1240K(5632K), 0.0020249 secs]
[GC (Allocation Failure) 2264K->1280K(5632K), 0.0007983 secs]
[GC (Allocation Failure) 2304K->1288K(5632K), 0.0009195 secs]
[GC (Allocation Failure) 2312K->1320K(5632K), 0.0016358 secs]
[GC (Allocation Failure) 2344K->1417K(5632K), 0.0015501 secs]
[GC (Allocation Failure) 2441K->1353K(5632K), 0.0008891 secs]
[GC (Allocation Failure) 2377K->1449K(5632K), 0.0005889 secs]
```

这一次使用 `-XX:+DoEscapeAnalysis` 选项开启了逃逸分析(其实该选项在 JDK 8 中默认是开启的，因此不需设置该选项)，JVM 一共只进行了 8 次 GC 回收，相比关闭逃逸分析时，次数少了很多。开启逃逸分析之后，JVM 之所以仍然存在 GC，是因为一开始运行时的循环次数不够，未触发 JIT，因此 Test 类实例对象仍然分配在堆空间，而当触发 JIT，JVM 使用 JIT 编译后的本地机器指令来实现类对象实例分配时，实例对象不再分配在堆上，因此不再有 GC 输出日志。

10.5.2 TLAB

相信很多研究 JVM 的道友对 TLAB 不会陌生，TLAB 的全称是 Thread Local Allocation Buffer，即线程本地分配缓存区，这是一个线程专用的内存分配区域。

TLAB 的出现能够解决直接在堆上安全分配所带来的线程同步性能消耗问题。堆内存是全

局的，任何线程都能够在堆上申请空间，因此每次申请堆内存空间时都必须进行同步处理。对于一个生产环境下的 Java Web 应用程序而言，拥有一两千、两三千个线程是很常见的事情，这么多线程同时在堆上申请内存，竞争十分激烈，必然会出现线程阻塞。而 TLAB 则是线程私有的一块内存空间，这块空间位于 eden 区。由于各个线程所拥有的 TLAB 区域彼此不重复，因此线程在各自的 TLAB 内存区域申请空间，无须加锁，这样内存申请的效率便会得到极大提升。其实说白了，这就是一种典型的“空间换时间”的策略。

参数-XX:+UseTLAB 可以开启 TLAB，默认是开启的。TLAB 的内存空间非常小，默认情况下仅占整个 eden 空间的 1%，每个线程所能拥有的 TLAB 空间非常少，因此 JVM 必然会限制 Java 类对象实例的大小。如果对象实例超过一定的阈值，便属于大对象，JVM 会将大对象直接分配在堆上，不再分配在 TLAB 区，否则一下子就可能将 TLAB 区“打爆”。

在上面贴出来的 BytecodeInterpreter::run() 函数源码中，可以看到 TLAB 的逻辑，其逻辑是直接调用 THREAD->tlab().allocate(obj_size)来完成 TLAB 内存分配。tlab 实际上是 JVM 内部 Thread 类的一个成员变量，类型是 ThreadLocalAllocBuffer，该类内部主要通过 3 个字段维护 TLAB 区域的范围，这 3 个字段分别是：

- ◎ _start，TLAB 区域的内存首地址。
- ◎ _top，最近一次 TLAB 分配内存后所指向的地址。
- ◎ _end，TLAB 区域的终止位置（内存对齐后的位置）。

通过这 3 个变量，TLAB 便能完成内存申请与释放。在 BytecodeInterpreter::run() 函数中调用 tlab->allocate(size_t)来申请内存，实现如下：

清单：/src/share/vm/memory/ThreadLocalAllocBuffer.inline.hpp

功能：TLAB 内存分配

```
inline HeapWord* ThreadLocalAllocBuffer::allocate(size_t size) {
    invariants();

    // 获取当前 top
    HeapWord* obj = top();
    if (pointer_delta(end(), obj) >= size) {

        // 重置 top
        set_top(obj + size);

        invariants();
        return obj;
    }
    return NULL;
}
```

这里的实现极为简单,如果 TLAB 剩余的空间足够容纳 Java 类对象实例,则重置 TLAB 的 `top` 属性,如此便完成内存分配。虽然 TLAB 分配的逻辑很简单,但是 TLAB 内存空间的维护稍具复杂性,需要随着运行时的执行流而随时变化。如果 TLAB 剩余空间不够,则 TLAB 内存分配必定会失败,此时 JVM 便会向 eden 区申请内存空间。如果 JVM 再次申请分配一个比较小的 Java 对象实例,该对象大小小于 TLAB 区的剩余空间,则 JVM 会继续将小对象优先填充进 TLAB 区域中。

如果 TLAB 区域都用完了,如何处理呢?别忘了,TLAB 区域本身被分配在 eden 区,属于 eden 区的一部分,因此当 eden 区也快要用完的时候,会触发 GC 垃圾回收,在垃圾回收期间,TLAB 的内存空间会随着 eden 区的回收一起被回收掉,如此实现 TLAB 的循环利用。

10.5.3 指针碰撞与 eden 区分配

如果 JVM 向 TLAB 申请内存失败,则会转而向 eden 区申请内存。在这个过程中,使用了 bump-the-pointer 技术,也即指针碰撞。其逻辑实现其实比较简单,就在上文贴出来的 `BytecodeInterpreter::run()` 函数源码中,其实现如下:

清单: `BytecodeInterpreter::run()`

作用: 指针碰撞

```
HeapWord* compare_to = *Universe::heap()->top_addr();
HeapWord* new_top = compare_to + obj_size;
if (new_top <= *Universe::heap()->end_addr()) {
    if (Atomic::cmpxchg_ptr(new_top, Universe::heap()->top_addr(),
        compare_to) != compare_to) {
        goto retry;
    }
    result = (oop) compare_to;
}
```

`Universe::heap()` 返回 JVM 内部所使用的 `CollectedHeap` 堆对象, `top_addr()` 指向 eden 区空闲块的起始地址, `end_addr()` 指向 eden 区空闲块的结束地址。JVM 首先通过 `compare_to` 保存 eden 区空闲块的起始地址,接着使用 `new_top` 保存分配内存后新的空闲块的起始地址。指针碰撞技术的关键在于 CAS 操作,这里通过基于 CPU 硬件的 CAS 原子指令进行空闲块的同步操作,比较 `_top` 的预期值与 `compare_to` 是否相同,若相同则表明没有其他线程操作该变量,若没有其他线程操作该变量,则会更新 eden 区的 `_top` 属性,并返回原来的 `_top` 作为 Java 类实例对象的内存首地址。这便是所谓的“指针碰撞”技术,其实就是判断预期的 `top` 与原来的 `top` 是否相等。而指针碰撞的关键就是 CAS 原语, JVM 通过 CAS 避免了多线程之间的锁竞争,这是实现内存快速分配的技术保障。

10.5.4 清零

前面两个快速分配策略——先在 TLAB 上分配，如果分配失败，则再向 eden 区申请内存空间，如果这两种分配策略中有一个成功，则 Java 类实例对象将会在 TLAB 区或者 eden 区占有一席之地。但是别忘了，无论是 TLAB 区还是 eden 区，都会不断地被 GC，因此 Java 对象实例所分配到的内存空间有可能仍残留着那些已经被回收或者被转移到其他堆内存区域的对象的信息片段，如果是这样，则需要将这段内存空间进行清零。在 `BytecodeInterpreter::run()` 函数源码中实现了清零逻辑：

清单：BytecodeInterpreter::run()

作用：清零

```
bool need_zero = !ZeroTLAB;
if (UseTLAB) {
    result = (oop) THREAD->tlab().allocate(obj_size);
}
if (result == NULL) {
    need_zero = true;

    // 向 eden 区申请内存
    // ....
}
if (result != NULL) {
    if (need_zero) {
        HeapWord* to_zero = (HeapWord*) result + sizeof(oopDesc) / oopSize;
        obj_size -= sizeof(oopDesc) / oopSize;
        if (obj_size > 0) {
            memset(to_zero, 0, obj_size * HeapWordSize);
        }
    }
}
```

在这段逻辑中，主要关注 `need_zero`，如果在 TLAB 区中成功分配内存，并且 TLAB 区本身已清零，则表达式 `!ZeroTLAB` 返回 `false`，于是 `if(need_zero)` 不满足条件，无需清零。否则，即使在 TLAB 区成功申请内存，最终仍需清零。如果 TLAB 区申请内存失败，则向 eden 区分配内存之前先将 `need_zero` 设置为 `true`，这意味着只要是从 eden 区分配的内存，最终都需要清零。清零的方式很简单，将指定的内存区域全部设置为 0 即可，所有的二进制位都是 0。

10.5.5 偏向锁

如果快速分配策略成功实施并完成清零，接着会设置偏向锁。所谓设置偏向锁，其实是设置对象头，即 oop 的 mark 标记，其逻辑如下：

清单: BytecodeInterpreter::run()

作用: 设置偏向锁

```
if (UseBiasedLocking) {
    result->set_mark(ik->prototype_header());
} else {
    result->set_mark(markOopDesc::prototype());
}
```

不是说好设置偏向锁的吗? 可是这里的源码看起来像是在设置 prototype。其实 prototype 的类型便是 mark, 而每一个 Java 类实例都有一个 mark 标记, 在前文讲解 oop-klass 这种面向对象的表达机制时曾提到, 所谓的 mark, 看起来像个 C++ 对象, 但实际上在 JVM 内部是被当作一个指针使用的, 在 32 位平台上, 指针就是一个 32 位的正整数, 同理, 64 位上的指针便是一个 64 位的正整数。而 JVM 会将 Java 类对象的 GC 分代年龄、哈希码、锁标志位等信息存放到这个 mark 上, 其实说白了就是二进制打标。其中, 偏向锁的标识也会打在这个 mark 指针上。

看上面这段源码, 如果 JVM 开启偏向锁, 则将 ik->prototype_header() 设置为新创建的 Java 类实例对象的标记。ik->prototype_header() 返回的标记中的偏向锁, 其实指向当前线程 ID, 而当前线程便是正在执行 new 指令、创建目标 Java 类对象实例的线程, 因此通过 result->set_mark(ik->prototype_header()) 便将该新创建的对象偏向锁偏向于当前创建它的线程, 如果在接下来的执行过程中, 该锁没有被其他的线程获取, 则持有偏向锁的线程将永远不需要再进行同步。

而如果 JVM 没有开启偏向锁, 则将 markOopDesc::prototype() 设置为新创建的 Java 类实例对象的标记。markOopDesc::prototype() 在前文也分析过, 其返回一个没有哈希码、没有偏向锁的标记。

关于偏向锁, 其概念往往与轻量级锁、重量级锁紧密关联在一起, 这几种锁可以相互转化。很多书籍与网络博客都有详细介绍, 感兴趣的道友可以深入研究。关于多线程同步控制是一个很复杂的话题, 从硬件到软件有各种各样的解决方案, 想要描述清楚, 本身便可以写一本书。如果感兴趣的道友非常多, 则笔者会在后续版本中详细讲解, 配合着 JVM 的 GC 机制一起深入分析。

10.5.6 压栈与取指

在快速分配流程走完偏向锁设置之后, Java 类实例对象的内存空间已经分配完成, 接着 JVM 将 Java 对象实例的内存首地址压入操作数栈栈顶, 完成压栈之后则开始取指——读取下一条字节码指令, 在 BytecodeInterpreter::run() 函数中实现了这种逻辑:


```
SET_STACK_OBJECT(result, 0); //压栈
UPDATE_PC_AND_TOS_AND_CONTINUE(3, 1); //取指
```

不过按道理，完成对象内存分配之后，不是应该将对象的内存首地址存储到局部变量表中对应的位置吗？这里为何只进行了压栈？道理很简单，Java 源码中的 `new` 语句通常会被编译为几条字节码指令，其中第一条字节码指令便是 `new` 指令，上面的逻辑都是 `new` 指令的内部实现机制。完成 `new` 指令之后，JVM 接着会调用 Java 类的构造函数，通过构造函数才真正返回一个完成原始构建的内部对象。构造函数运行之后，则会生成一条字节码指令，将对象的内存首地址存储到局部变量表中。例如下面的示例：

清单：Test.java

功能：new 指令

```
public class Test {
    public void test(){
        Test t = new Test();
    }
}
```

该示例特别简单，编译后，使用 `javap` 命令查看其字节码指令，如下：

```
public void test();
  descriptor: ()V
  flags: ACC_PUBLIC
  Code:
    stack=2, locals=2, args_size=1
       0: new           #2             // class Test
       3: dup
       4: invokespecial #3             // Method "<init>":()V
       7: astore_1
       8: return
```

可以看到，源码中的 `new` 语句对应了 4 条字节码指令，首先执行 `new` 指令，接着执行 `invokespecial` 指令调用类的默认无参构造函数 `<init>()`，最后再通过 `astore_1` 将构造好的实例对象内存首地址保存进局部变量表中。由此可以看出，完成 `new` 指令后，之所以不立即将对象内存地址写入局部变量表中，是因为接下来就会调用方法，而 JVM 每次调用 Java 方法之前，都必须要将入参压入操作数栈栈顶。如果执行完 `new` 指令之后就立即将对象内存地址写入局部变量表中，那么接下来调用类的构造函数时就需要再次将对象内存地址从局部变量表中读取出来并压入操作数栈栈顶，这样多了几次内存读写，所以 JVM 干脆就在执行完 `new` 指令后，直接将内存地址压入栈顶，提升性能。

上面分析了 Java 类对象内存的快速分配的技术实现机制，总体而言，快速分配的策略示意图如图 10.6 所示。

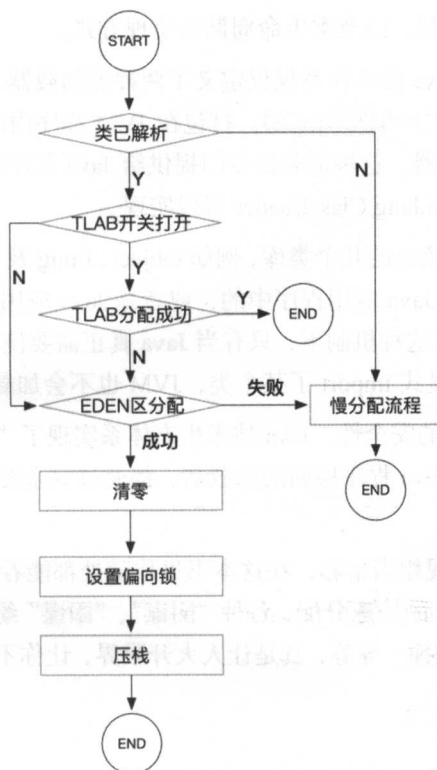


图 10.6 Java 类实例对象内存快速分配流程

如果快速分配失败,则最终会进入慢分配流程,慢分配流程也会首先尝试在 TLAB 中分配,如果分配失败,则继续尝试使用指针碰撞技术在新生代分配,这种分配是无锁的,效率仍然很高。如果仍然分配失败,则最终才会使用互斥锁,在堆区进行分配。在这个过程中如果碰到 GC 正在回收垃圾,则会等待 GC 回收完成。慢分配流程中所使用的优化手段与快速分配类似,都是优先使用无锁分配方案(TLAB 或指针碰撞)。慢分配与 GC 在理论和源码上是紧密耦合在一起的,而 GC 本身是一个非常复杂的管理体系,本书限于篇幅不展开细讲,如果有很多道友都对此感兴趣,我们可以在后续版本中继续交流。

10.6 本章总结

类的加载机制与生命周期等概念,在各种书籍与各种网络博客里随处可见,然而对于一个想要真正了解其内部实现的人而言,那些都涉入过浅。本章“拨云见日”,从 JVM 源码的角度,

还原出 Java 类加载的真实机制，以及类生命周期的实现方式。

从程序的角度看，其实 Java 技术体系仅仅定义了两种类加载器——一种是 boot class loader，即引导类加载器，其是使用 C++ 编写而成的，打包在 JVM 程序内部。另一种则是 JDK 类库中的 `java.lang.ClassLoader` 加载器，这种加载器专门提供给 Java 程序使用，所有在 Java 程序中定义的种类加载器最终都通过 `java.lang.ClassLoader` 得以实现。

当 JVM 启动时，会加载核心的几个类库，例如 `Object`、`Long` 及 `Integer` 等。剩下的 Java 类，无论是 JDK 类库中的，还是 Java 应用程序中的，或者是 Java 应用程序所依赖的第三方 jar 包，都采用“延迟加载”机制。在这种机制下，只有当 Java 真正需要使用某个类时，JVM 才会真正加载，否则，即便在程序中显式 `import` 了某个类，JVM 也不会加载。

为了确保 JDK 核心类库的安全性，Java 技术生态体系实现了“双亲委派”机制，无论是内部的 boot class loader，还是 Java 程序层面的加载器，都通过双亲委派机制，保护核心类不被开发者破坏和伪造。

类的实例分配的技术实现相当精彩，在这本书里你随处都能看到那些伟大工程师们的思想闪光点。为了实现类实例内存的快速分配，各种“阴谋”、“阳谋”纷纷上场，TLAB 无锁技术、栈上分配、标量替换、指针碰撞，等等，真是让人大开眼界，让你不禁感慨，这才是把技术“玩到了家”。

名家好评

Java 从 1995 年发布以后,已经发展成为一门流行的编程语言。业界也有无数的书介绍 Java 语言的方方面面。但是,本书不仅讲解了 Java 虚拟机的内部实现机制,还深入分析了为什么要这么实现。每一种技术设计的背后,都有其必然性。能够知其然,并知其所以然,才能透过现象看本质,举一反三,实现技术升华。亚飞有多年的 Java 实践,尤其是在菜鸟网络,需要具备处理高并发、大型工程的架构经验,相信他在书中会有更多的视角分享给读者。

——王文彬(菲青),菜鸟 CTO

作者凭借深厚的 C 与 Java 技术功底以及多年对于 JVM 的深入研究编写的这本书,真正从虚拟机指令执行处理层面,结合 JVM 规范的设计原理,完整和详尽地阐述了 Java 虚拟机在处理类、方法和代码时的设计和实现细节。同时书中大量的代码和指令细节能够让程序员更加直接地理解相关原理。

这是一本优秀的技术工具书,可以让阅读者更加深刻地理解虚拟机的原理和细节,值得每一位具有极客精神、追求细节的优秀程序员反复阅读和收藏。

——陌铭,菜鸟平台技术部架构师

本书是一本通过深入结合 HotSpot 源代码来解释 Java 虚拟机工作机理的书籍。概念是一切知识结构的基石。通过阅读本书,Java 工程师可以了解和掌握 Java 虚拟机的核心概念,可以领会在工作中如何开发合理的、高效的 Java 应用,如何有效地解决、排查 Java 问题。

——李三红,阿里巴巴/蚂蚁金服 JVM 架构师

作为一名 Java 程序员,我们写过很多 Java 程序。但是,Java 程序到底是如何运行的?如何写出更高效的 Java 代码……? 这些问题不仅是初学者的困惑,也是很多老司机的短板。这些问题归根结底都要从 JVM 中寻找答案,而国内能深入分析 JVM 的书并不多。亚飞撰写的本书不仅深入分析了 Java 虚拟机的运行机制与原理,而且在表达上非常通俗易懂,可以帮助读者深入理解并掌握 Java 语言的核心细节,在开发工作中以不变应万变,写出优秀高效的代码,值得细读。

——许令波(君山),《深入分析 Java Web 技术内幕(修订版)》作者

作者简介

封亚飞,任职于菜鸟物流云平台,负责中间件开发。

注册博文视点社区(www.broadview.com.cn)用户,即享受以下服务:

! 提勘误赚积分:可在【提交勘误】处提交对内容的修改意见,若被采纳将获赠博文视点社区积分(可用来抵扣购买电子书的相应金额)。

! 交流学习:在页面下方【读者评论】处留下您的疑问或观点,与作者和其他读者共同交流。

页面入口: <http://www.broadview.com.cn/31541>



策划编辑:刘 皎
责任编辑:郑柳洁
封面设计:吴海燕

欢迎投稿

邮箱: Ljiao@phei.com.cn
电话: 010-88254395
新浪微博: @皎丫子

上架建议: 编程语言-JAVA

ISBN 978-7-121-31541-1



9 787121 315411 >

定价: 129.00元